Agents and Objects: An Empirical Study on Software Engineering

Alessandro Fabricio Garcia | Cláudio Nogueira Sant'Anna | Christina von Flach Garcia Chavez
Viviane Torres da Silva | Carlos José Pereira de Lucena | Arndt von Staa

Computer Science Department – SoCAgents/TecComm Group
Pontifical Catholic University of Rio de Janeiro – PUC-Rio
Rua Marquês de São Vicente, 225 – Ed. Pe. Leonel Franca, 10º Andar
Rio de Janeiro – Brazil

{afgarcia,claudios,flach,viviane,lucena,arndt}@inf.puc-rio.br

PUC-RioInf.MCC06/03 Fevereiro, 2003

Abstract. With multi-agent systems (MASs) growing in size and complexity, the separation of their concerns throughout the different development phases is a main need for MAS engineers. Separation of concerns is a well-known principle in software engineering to achieve improved reusability and maintainability of complex software. Hence it is necessary to investigate systematically whether established and evolving abstractions from Software Engineering are able to support the explicit separation of MAS concerns. This paper presents an empirical study that compares the maintenance and reuse support provided by abstractions associated with two OO techniques for MAS development: aspect-oriented development and pattern-oriented development. The gathered results have shown that the abstractions from the aspect-oriented approach allowed the construction of a MAS with improved separation of MAS concerns. Also, the use of this approach resulted in: (i) less lines of code, (ii) less components, (iii) lower component cohesion, and (iv) lower coupling between the components. An additional important finding of this empirical study is that the aspect-oriented approach also supported a better alignment with higher-level abstractions from agent-oriented models.

Keywords: Separation of concerns, multi-agent systems, empirical software engineering, metrics, aspects, design patterns.

Resumo. Com o crescimento do tamanho e complexidade de sistemas multi-agentes (SMAs), a separação dos seus *concerns* através das diferentes fases de desenvolvimento passou a ser uma das principais necessidades dos engenheiros de SMAs. Separação de *concerns* é um princípio conhecido de engenharia de software para melhorar a reusabilidade e manutenibilidade de software. Portanto, é necessário investigar sistematicamente se as abstrações de engenharia de software já estabelecidas, e também aquelas que ainda estão em emergência, são capazes de suportar a separação explícita de *concerns* de SMAs. Este artigo apresenta um estudo empírico que compara o suporte à manutenção e ao reuso obtido com as abstrações associadas a duas técnicas OO para o desenvolvimento de SMAs: desenvolvimento orientado a aspectos e desenvolvimento orientado a padrões. Os resultados obtidos mostraram que as abstrações da abordagem orientada a aspectos permitiram a construção de um SMA com melhor separação de *concerns*. Além disso, o uso dessa abordagem trouxe os seguintes resultados: (i) menos linhas de código, (ii) menos componentes, (iii) menor coesão dos componentes e (iv) menor acoplamento entre os componentes. Com este estudo empírico, pode-se verificar também que a abordagem orientada a aspectos suportou um melhor alinhamento com as abstrações dos modelos orientados a agentes.

Palavras-chave: Separação de *concerns*, sistemas multi-agentes, engenharia de software experimental, métricas, aspectos, padrões de projeto.

1. Introduction

As agent-based software engineering evolves there is a need for better understanding of the relationships between its abstractions and concerns and the ones from object-orientation. Large-scale MASs involve complex and non-orthogonal concerns, such as agent types, autonomy, adaptation, interaction, collaboration, roles and so forth. The analysis and design of MAS concerns are directly supported by a growing number of abstractions associated with agent-oriented languages and methodologies [20]. Software engineers in turn however mostly rely their development phases based on OO design techniques and programming languages, such as Java (Figure 1). This transition can be cumbersome not only because the set of abstractions is different in the generated artifacts, but also because most recurrent concerns of MAS applications are essentially different from those of OO systems. MAS concerns may not be explicitly separated by existing OO abstractions. Therefore it remains to be verified if they lead to the production of MAS components, which are lowly coupled, highly cohesive, ease to understand, evolve and reuse.

However, no systematic study has been conducted so far to establish whether the MAS concerns can be naturally supported by well-known abstractions. Besides few empirical studies in the literature have investigated the interplay between agent-oriented abstractions and OO pure and emerging abstractions. Research on agent-based software engineering has primarily focused on the development of agent-oriented methodologies and modeling languages, without focusing on such interplay [16]. Many researchers (such as [21, 38]) have argued persuasively that the concerns associated with MASs are often much different from those traditionally associated with OO systems; and hence the OO abstractions generally fail to capture the relevant concerns of MASs. They also argue that it is not possible to consider the co-existence of agents and objects because they are essentially different. However, such statements originate mainly from experts' opinions and assertions based on informal experience but are not a result of empirical evidence.

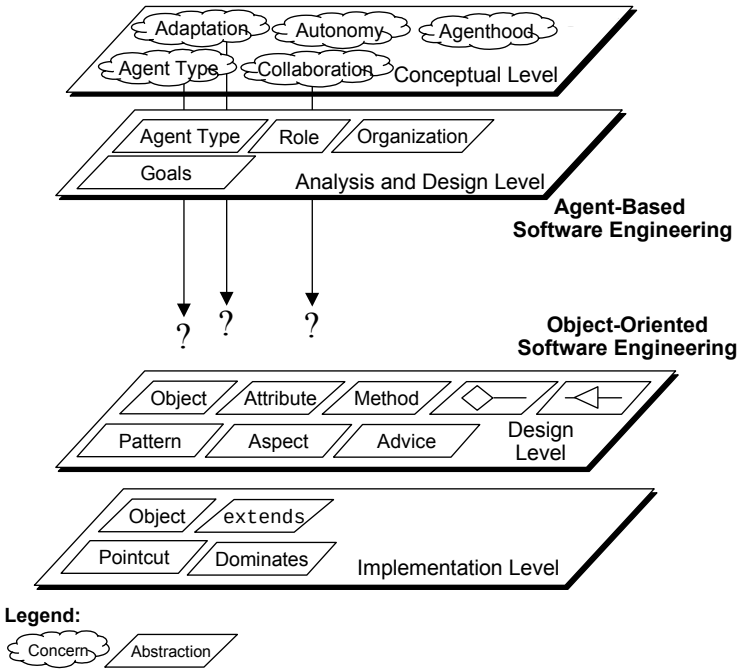


Figure 1. Misalignments and Discontinuity in Agent-Based Software Engineering.

Experimental studies [2] are the most effective way to supply empirical evidence that may improve our understanding about software engineering. In this context, this paper presents an experiment that makes systematic use of two different OO techniques for MAS development. The overall goal of this study is to evaluate the maintenance and reuse support of the investigated techniques for addressing MAS concerns. An extended set of traditional metrics based on established software engineering principles was applied to provide software engineers with a better understanding of the interplay between OO abstractions and agent-oriented ones. The gathered results provide a clear understanding of the strengths and weaknesses of the two investigated techniques and their compatibility and divergences. The results are crucial for the development and potential convergence of OO and agent-oriented techniques. They are also useful for engineers of realistic MASs who need to implement the agent-oriented models using OO programming languages. The conclusions may also be of interest to agent-oriented methodologists since they may decide to incorporate solutions for problems detected in our study directly as part of their methodologies.

Both aspect-based and pattern-based techniques are used in our work to design and implement a MAS that supports virtual development environment systems. The evaluation of the two systems is conducted by using a proposed quality model and Basili's GQM methodology [3] based on which our metric suite is identified. Several scenarios generated during the case study are used to evaluate the reuse and maintainability of MASs. The remainder of this paper is organized as follows. Section 2 presents the investigated techniques and the associated methods. Section 3 discusses the study organization in terms of the methodology used, associated goals and questions, stated hypotheses, subjects involved and the used project. Section 4 presents our evaluation framework which is composed of a quality model, related metrics and a set of scenarios. Section 5 presents the study results, which are interpreted and discussed in Section 6 based on the stated hypotheses. Section 7 includes some concluding remarks and directions for future work.

2. The Investigated Techniques

As stated previously, we have selected two techniques for MAS development: pattern-oriented development [7, 14] and aspect-oriented development [25, 37]. We have specially selected these techniques since they extend the basic set of OO abstractions to promote improved separation of concerns. Further, their additional abstractions (such as aspects and patterns) have been proposed as promising solutions to reduce coupling, increase cohesion, and promote software reuse and maintenance.

Each technique uses a different set of different abstractions which are illustrated in Table 1. The abstractions are classified into three kinds: components, component elements, and relationships. Note that the set of abstractions used by each technique is not disjoint. Both techniques explore the classical abstractions from OO software engineering such as classes, objects, inheritance, association, and so on. The basic difference is that the first one is centered on the use of the pattern abstraction and the second one is centered on the use of the aspect abstraction. The first line of Table 1 describes the main abstractions of agent-based software engineering.

	ABSTRACTIONS			
	Central Abstractions	**Components**	**Component Elements**	**Relationships**
AOSE	Agents	Agents, Roles, Organizations, Environments…	Beliefs, Goals, Actions, Plans, Commitments…	Play, Control, Aggregation, …
Pattern-Oriented Technique	Patterns	Classes, Objects, Interfaces, Abstract Classes	Attributes, Methods	Inheritance, Association, Aggregation, …
	Classes			
Aspect-Oriented Technique	Aspects	Aspects, Aspect Instances, Abstract Aspects	Advices, Pointcuts Joinpoints	Crosscutting

Table 1. The abstractions associated with the investigated techniques

Since the investigated techniques were not developed with MAS concerns (or *agency concerns*) in mind, two supporting methods [15, 18], specially tailored to the MAS development, were associated with each technique and used by the experiment subjects to apply the respective technique and associated abstractions. The objective of these methods is threefold:

(1) promote separation of agency concerns based on abstractions of the respective technique;

(2) minimize the misalignments between agent-based high-level models and OO design and implementation;

(3) support the construction of reusable and maintainable large MAS.

Each method step achieves separation of MAS concerns through the isolation of the concerns in individual abstractions or through a set of interrelated abstractions. The methods minimize the misalignments and covers gaps between abstractions from agent-based modeling and OO design by indicating which OO abstraction(s) to use for given abstractions from high-level agent models. Both methods are independent of MAS implementation frameworks, like JADE [4], ZEUS [29], and Retsina [35], and specific communication language, such as ACL [1] and KQML [13]. This independence is important for the scope of this work since we are not focused on particularities of implementation frameworks or communication languages, but on advantages and limitations of pure and emerging OO abstractions to support MAS development. The subsections below summarize the relevant features of the techniques and respective methods. The full description of the techniques [14, 25, 37] and methods [15, 18] is reported elsewhere since it is out of the scope of this paper.

2.1. Pattern-Oriented Development

Pattern-oriented (PO) development [7, 14] is a software engineering technique concerned with the application of design patterns. A pattern provides a design and implementation solution to a recurring problem, defining a set of components and their relationships [7]. The use of design patterns has been greatly advocated in OO software development, with emphasis on reuse and maintenance. The pattern solution structures and disciplines the composition of the separated components, ensuring that the system can only change or evolve in specific, predictable ways. Patterns are the building blocks of large-scale software systems, which are likely to include instances of more than one of these patterns, composed in arbitrary ways. Design patterns are not restricted to the object paradigm, but in this work we focus on OO

design patterns. In the MAS context, a pattern can be widely used as a design and implementation metaphor for recurrent OO structures for MASs, and thus minimizing the misalignments between high-level agent designs and their OO detailed design and implementation.

The Abstractions. The abstractions associated with the pattern-oriented technique are as follows. The basic *components*: are classes, objects, interfaces, abstract classes, and the patterns themselves. A pattern itself can be viewed as an abstraction since it encapsulates a solution in terms of components, their internal elements and relationships, to a recurrent problem in the development of MAS. There are two *elements* internal to these components: attributes to represent the component state, and methods to represent the component operations. The basic component *relationships* are inheritance, association, and aggregation.

Separation of MAS Concerns. Separation of concerns can be achieved through the isolation of the concerns of interest into individual classes or a set of cohesive interrelated classes. The pattern-oriented method supports separation of MAS concerns by explicitly associating MAS concerns with a set of suitable abstractions from pattern-oriented development. Each method step indicates how to structure a given MAS concern. Figure 2 shows some steps of the pattern-oriented method. For example, the first step guides the structuring of the *agenthood* concern in terms of a set of classes, methods, and attributes. In the sequence, the second step deals with the *agent types* concern, the third step is related to the basic agency properties, such as *adaptation*, *interaction* and *autonomy* and the separation between them, the *collaboration* concern is represented by the composition of the mediator pattern [14] and the role pattern, and so forth. This separation of MAS concerns are essential to MAS engineers since they can decide to extended and modify such concerns as the system evolves.

Traceability and Gap Covering. The pattern-oriented method minimizes the misalignments and covers gaps between abstractions from agent-based modeling and object-oriented design by indicating which abstration(s) to use for given abstractions from agent-based software engineering. For example, agents are represented as classes, roles are represented using the role pattern, plans are represented as methods, and agent knowledge is represented as attributes. Both general patterns, such as Mediator and Composite patterns, and MAS-specific patterns [22, 34] are used to address the traceability between agent-oriented models and object-oriented designs.

Reuse and Maintenance Support. The reuse and maintenance support is a natural consequence of the improved separation of MAS concerns and the traceability between agent-oriented abstractions and pattern-oriented abstractions, both provided by the pattern-oriented method. Moreover each pattern itself is intended to support future reuse and maintenance activities in the MAS. Design patterns offer solutions that structure and discipline the composition of the separated parts, ensuring that the system can only change or evolve in specific, predictable ways. The use of the patterns provides an additional abstraction level and its use can minimize the complexity caused by the presence of numerous conceptual differences between agent abstractions and object abstractions in large MASs. So the pattern-oriented solution provides a refined language to MAS designers discuss elements from the high-level agent models in terms of their implementation based on object-oriented abstractions. In short, each pattern is the foundation to promote construction ease, maintainability and reusability of MASs.

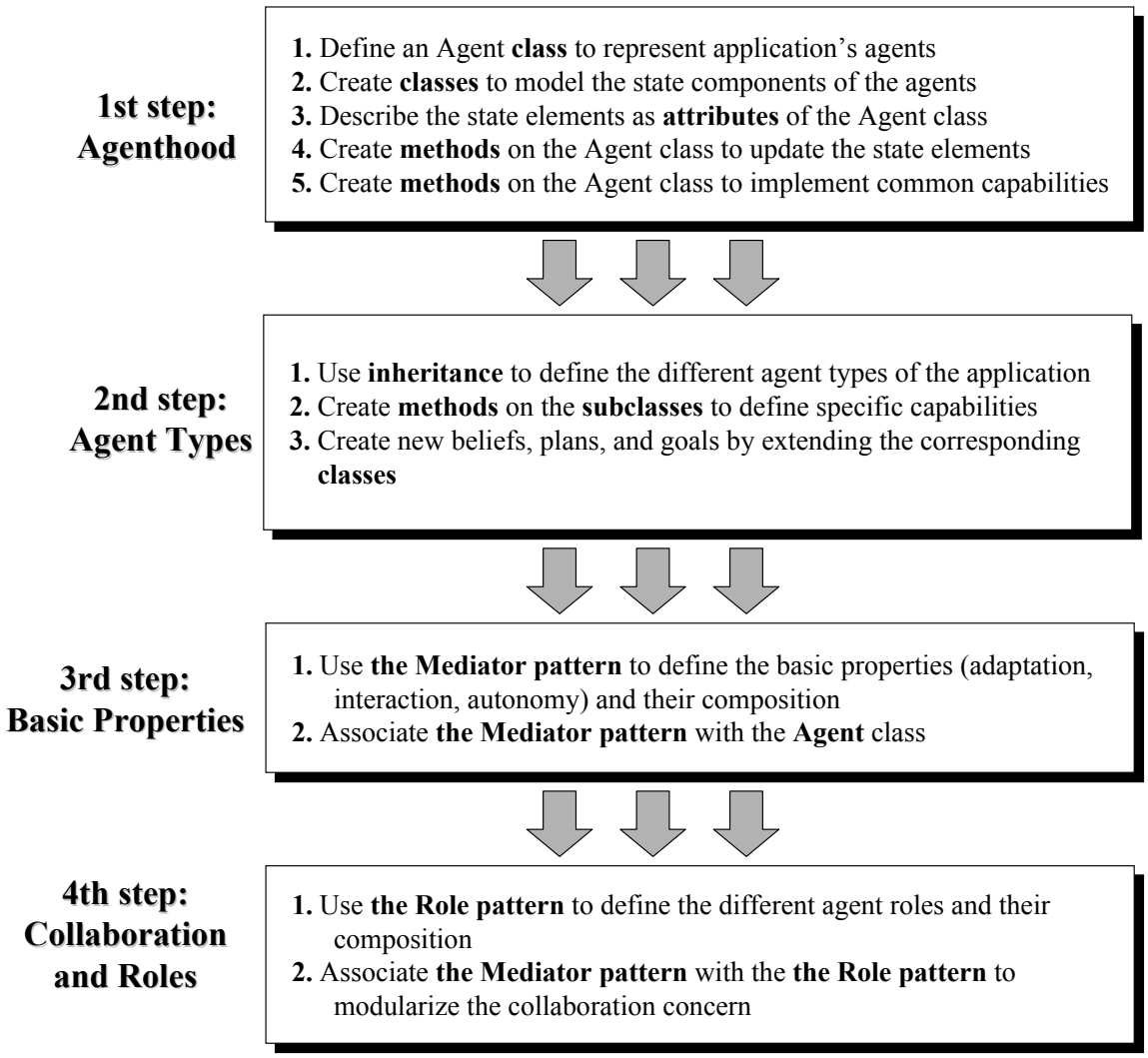


Figure 2. The Pattern-Oriented Method for MAS Development

2.2. Aspect-Oriented Development

Aspect-oriented (AO) software development [25, 37] has been proposed as a technique for improving separation of concerns in software construction and support improved reusability and maintainability. The aspect-oriented technique is not restricted to the object paradigm [25], but it is our focus in this experiment. The central idea is that while pure abstractions of the object paradigm are extremely useful, they inherently are unable to modularize all concerns of interest in complex systems. Thus, the goal of the AO technique is to support crosscutting concerns, by providing abstractions that make it possible to separate and compose them to produce the overall system. Crosscutting concerns are defined as system concerns that crosscut components in the design and implementation of a system.

Aspects are modular units of crosscutting concerns that are associated with a set of classes or objects. Central to the process of composing aspects and classes is the concept of *join points*, the elements that specify how classes and aspects are related. Join points are well-defined points in the structure and dynamic execution of a system. Examples of join points are method calls, method executions, and field

sets and reads. *Pointcuts* are collections of join points. *Advice* is a special method-like construct that can be attached to pointcuts. In this way, pointcuts are used in the definition of advices. There are different kinds of advices: (i) a *before advice* runs whenever a join point is reached and before the actual computation proceeds; (ii) an *after advice* runs after the computation "under the join point" finishes, i.e. after the method body has run, and just before control is returned to the caller; (iii) an *around advice* runs whenever a join point is reached, and has explicit control whether the computation under the join point is allowed to run at all. An aspect may also define attributes and methods to the classes to which the aspect is attached. Weaver is the mechanism responsible for composing the classes and aspects (Figure 3). AspectJ [26] is a practical aspect-oriented extension to the Java programming language. Up to the current version of AspectJ, almost all of the weaving process is realized as a pre-processing step at compile-time [26].

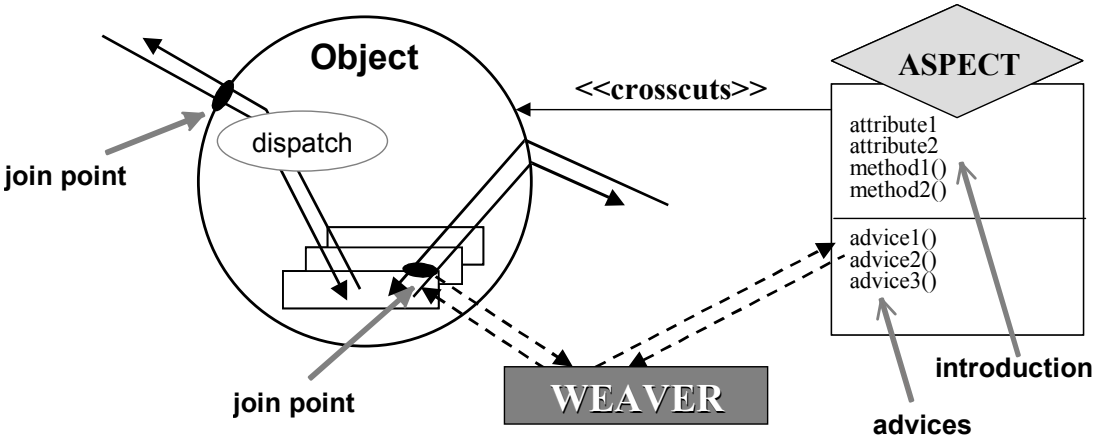


Figure 3. Abstractions for Dealing with Crosscutting Concerns.

The Abstractions. The abstractions associated with the aspect-oriented technique are described in the following. The basic *components*: in addition to the object-oriented components (classes, objects, interfaces, abstract classes), the technique deals with aspects, as described above. The *elements* internal to the classes, objects and interfaces are the same to the pattern-oriented technique: attributes to represent the component state, and methods to represent the component operations. In addition, the aspect elements include advices, pointcuts, joinpoints, attributes and methods. The basic component *relationships* are inheritance, association, and aggregation, as in the pattern-oriented technique. However, it also includes crosscutting which is a relationship established between a class and an aspect.

Separation of MAS Concerns. Separation of concerns can be achieved through the isolation of the concerns of interest into individual classes or aspects. The aspect-oriented method supports separation of MAS concerns by explicitly associating MAS concerns with a set of suitable abstractions from aspect-oriented development. Each method step indicates how to structure a given MAS concern. Figure 4 shows some steps of the aspect-oriented method. For example, the first step guides the structuring of the *agenthood* concern in terms of a set of classes, methods, and attributes. In the sequence, the second step deals with the *agent types* concern, the third step is related to the basic agency properties, such as *adaptation*, *interaction* and *autonomy* and the separation between them, the *collaboration* concern is represented by the composition of the collaboration aspect and the role aspects, and so forth. This

separation of MAS concerns are essential to MAS engineers since they can decide to extended and modify such concerns as the system evolves.

Traceability and Gap Covering. The aspect-oriented method minimizes the misalignments and covers gaps between abstractions from agent-based modeling and object-oriented design by indicating which abstration(s) to use for given abstractions from agent-based software engineering. For example, agents are represented as classes, roles are represented as aspects, plans are represented as methods, and agent knowledge is represented as attributes. The main difference between the aspect-oriented method and the pattern-oriented method is that the former explores aspects to represent roles and agent properties (such as adaptation, interaction, autonomy, collaboration, and so on), while the later uses known patterns to represent those concerns. The main idea is the use of aspects to encapsulate agency properties and roles and separate them from the agent's basic functionalities. The separation is very helpful to promote traceability among agent-oriented and aspect-oriented artifacts, since when an agent property or role is added or removed in the agent-oriented modeling, it is directly added or removed in the aspect-oriented design and code.

Reuse and Maintenance Support. The reuse and maintenance support is a natural consequence of the improved separation of MAS concerns and the traceability between agent-oriented abstractions and aspect-oriented abstractions, both provided by the aspect-oriented method. Moreover aspects are claimed to facilitate software construction and maintenance and to increase the potential for reuse, since they aim an improvement on software modularity. The aspects provide an additional abstraction level and its use can minimize the complexity caused by the presence of numerous conceptual differences between agent abstractions and object abstractions in large MASs. So the aspect-oriented solution provides a refined language to MAS designers discuss elements from the high-level agent models in terms of their implementation based on classes, aspects, and their relationships and internal elements. In short, each aspect is the foundation to promote the maintenance and reuse of the components of MASs.

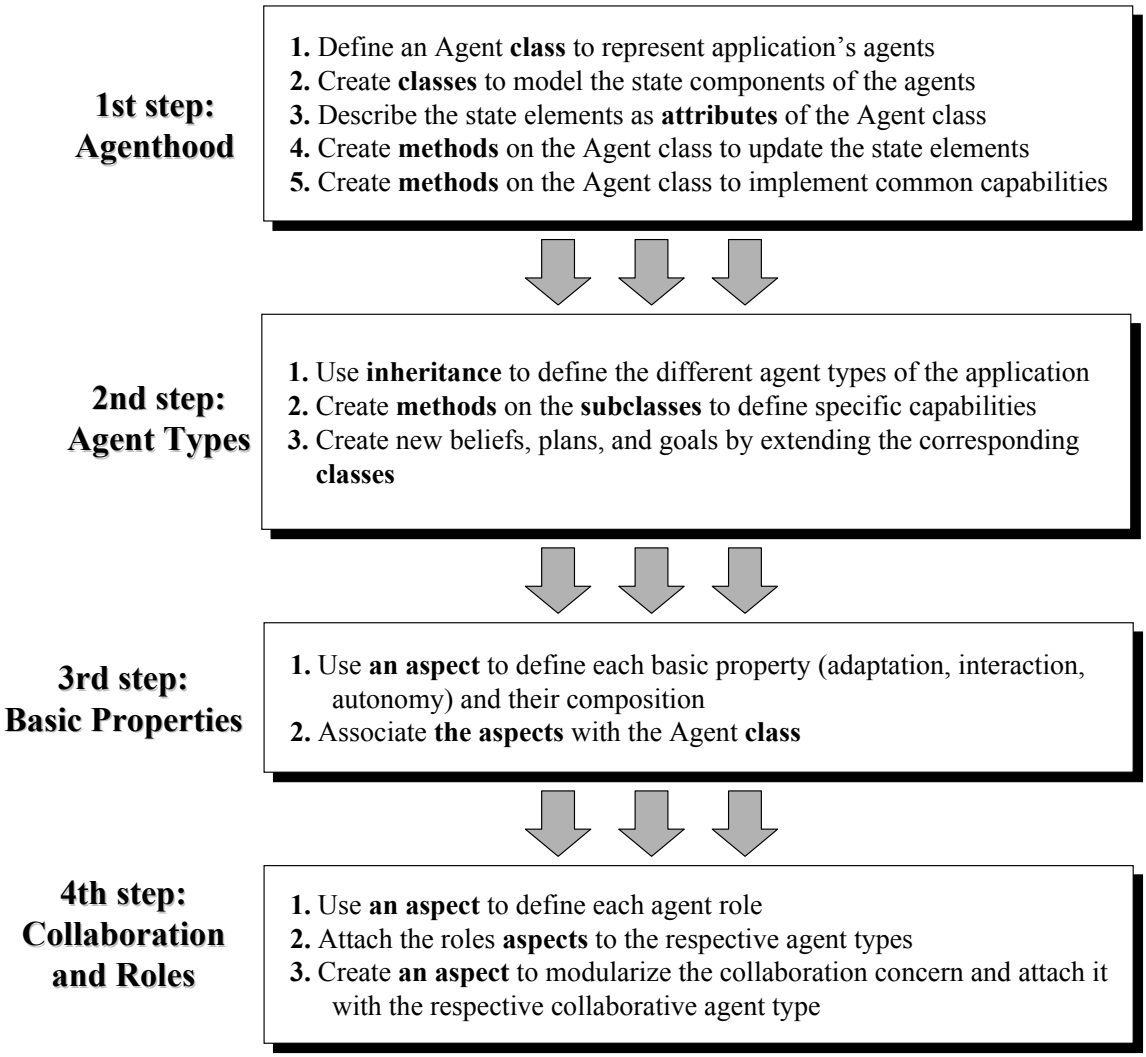


Figure 4. The Aspect-Oriented Method for MAS Development

3. The Empirical Study

3.1. The Methodology

The evaluation of software techniques is a notoriously hard task. There are very few established methodologies for measurement planing and data gathering. The experiment organization was based on a complementary application of the Basili's GQM (Goal/Question/Metric) methodology [3] and our evaluation framework (Section 4) which we defined specially to the context of this study. The GQM methodology was used to structure the experiment in terms of its goals, and the evaluation framework was defined to elicit the qualities, factors and criteria investigated in this experiment. Both were helpful to find existing metrics and define new ones for this empirical study. The GQM methodology was selected to evaluate the investigated techniques as it has gained widespread popularity and support within the software engineering community. The GQM paradigm has been proposed as a goal-oriented approach for the measurement of products and processes in software engineering. The GQM methodology is based upon the assumption that to gain a practical measure one must first understand and specify the goals of the software artifacts being measured, and the goals of the measuring process. The GQM approach provides a framework involving three steps:

1. List the major ***goals*** of the empirical study;
2. Derive from each goal the ***questions*** that must be answered to determine if the goals are met;
3. Decide what must be measured in order to be able to answer the questions adequately (definition of the ***metrics***).

Section 3.2 defines our goals and questions, while Section 4 presents the metrics as part of our evaluation framework.

3.2. Goals and Questions

The overall goal of this study is to evaluate the maintainability and reusability of the 2 investigated techniques in the MAS context. It also indirectly provides software engineers with a better understanding of the interplay between abstractions from agent-oriented analysis and design and the ones from OO design and implementation. This goal was refined into a set of questions, which represent an operational definition of them. The objective was to generate as many questions as possible, including redundant or invalid questions. As the process was continuing, we developed a hierarchical set of questions that were subsequently narrowed. For each question the relevant metrics were defined (Section 4.1.3). The generated framework is also useful when interpreting the data. Figure 5 presents a sample of the questions generated. The Appendices A presents the goals and questions that remained after refining.

3.3. Hypotheses

The hypotheses to be tested are stated as follows:

H1: the aspect-oriented technique provides better support for MAS maintainability and reusability;
H2: the abstractions from OO software engineering are not suitable for the design and implementation of MASs.

The first hypothesis is based on a previous qualitative study, which we have conducted previously [15]. It is related with the goal stated in the Section 3.2. The second one is influenced by the arguments of many MAS researchers [21, 38].

3.4. The MAS Project

The project upon which this system is based has been derived from a case study undertaken in the TecComm/SoC+Agents Group at PUC-Rio in Brazil (from herein referred to as Portalware). Portalware is a web-based environment that supports the development and management of Internet portals. As the needs of the Internet Portals market change ever more rapidly the frailties in the used software engineering techniques become increasingly apparent. To survive, Portalware must remain extensible and modifiable, and so its design and implementation must be capable of responding to change. The agent-oriented system modeling was based on Elammari's modeling language [10] and on TAO modeling framework [33]. TAO was particularly used because this elicits common abstractions used in MAS analysis and design. UML notations [5] and the Java language were respectively used to generate the pattern-oriented designs and implementation. An UML extension for aspect-oriented design [8] and the AspectJ programming language [26] were used to generate the aspect-oriented designs and implementation.

Goal

Evaluate the reusability and ease of evolution of the implemented multi-agent systems in order to compare the object-oriented development with the aspect-oriented development.

Questions

1. How easy is it to evolve the system?
    - 1.1. How easy is it to understand the system?
        - 1.1.1. How concise is the system?
            - 1.1.1.1. How many components are there?
            - 1.1.1.2. How many lines of code are there?
            - 1.1.1.3. How many attributes are there?
            - 1.1.1.4. How many methods and advices are there?
        - 1.1.2. How well are the agency concerns localized?
            - 1.1.2.1. How scattered is the agenthood definition?
            - 1.1.2.2. How scattered are the agent fundamental properties?
                - 1.1.2.2.1. How scattered is the autonomy property?
                - 1.1.2.2.2. How scattered is the interaction property?
                - 1.1.2.2.3. How scattered is the adaptation property?
            - 1.1.2.3. How scattered are the agent alternative properties?
                - 1.1.2.3.1. How scattered is the collaboration property?
            - 1.1.2.4. How scattered are the agent roles?
            - 1.1.2.5. How scattered is the definition of agent types and their instances?
                - 1.1.2.5.1. How scattered is the user agent type definition?
                - 1.1.2.5.2. How scattered is the definition of a user agent's instance?
                - 1.1.2.5.3. How scattered is the information agent type definition?
                - 1.1.2.5.4. How scattered is the definition of an information agent's instance?
        - 1.1.3. How high is the coupling of the system?
            - 1.1.3.1. How high is the coupling between components?
        - 1.1.4. How high is the cohesion of the system?
            - 1.1.4.1. How high is the cohesion of the system components?

Figure 5. An example of the questions generated by GQM

The MAS concerns handled in this project are that ones of real-world reactive MASs, typical of many existing applications. This MAS encompasses several agency concerns, including agent types, roles, collaboration, interaction, adaptation, autonomy, and so on. This environment includes a number of *agent types* to control portals, and to coordinate and automate the time-consuming repetitive activities of the development groups. Portalware encompasses 3 agent types: (i) *interface agents*, (ii) *middle agents*, and (iii) *user agents*. The agent types implement the *agenthood* concern and other concerns. In this system, the agenthood concern is comprised of three basic agency properties: *interaction*, *autonomy*, and *adaptation*. Each agent instance has different agency properties and play distinct *roles*. Middle agents are devoted to mediate the conversations between all system agents, providing services like naming service.

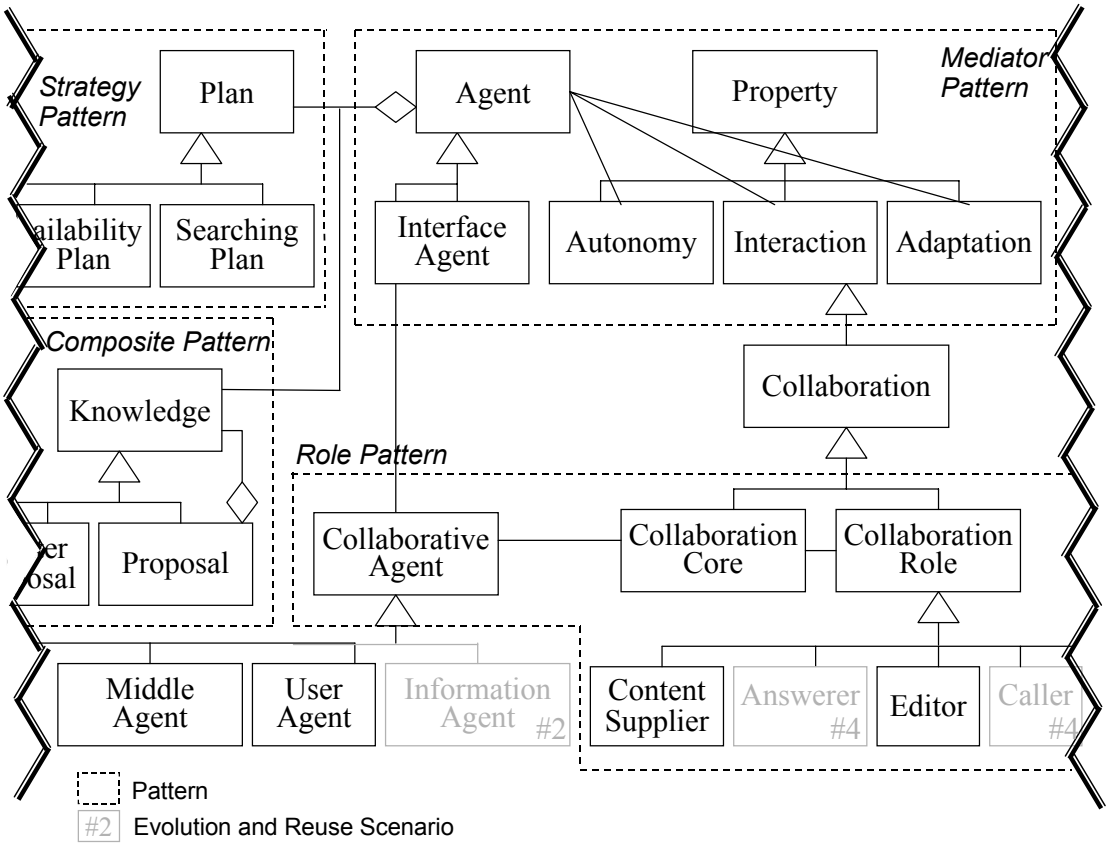


Figure 6. A Slice of the Pattern-Oriented Design of the MAS

User agents represent Portalware users and are implemented to reduce the need for cross talk between working users. Several roles are attributed to Portalware users and its respective agents, but the main ones are: (i) the *Content Supplier* role, and (ii) the *Editor* role. A content supplier (CS) is responsible for providing the portal with content segments (for instance, news). The editor is responsible for selecting from the available content segments for publishing and for assigning roles to the users. Also the agent playing an editor role has the responsibility to contact the prospective CSs and negotiate with them for the use of their capabilities. Since editors and CSs need to collaborate with each other to maintain the web portal, user agents incorporate *plans* for automating and supporting collaboration in different contexts. Interface agents monitor the graphical system interface in order to interact with Portalware users. They learn short cuts by capturing preferences, and by receiving explicit instructions from the user. For

instance, interface agents operate while a content supplier is authoring content segments, using keywords entered in the document and its acquired model of the user's preferences. Interface agents do not incorporate the *collaboration* concern since they do not cooperate with other software agents.

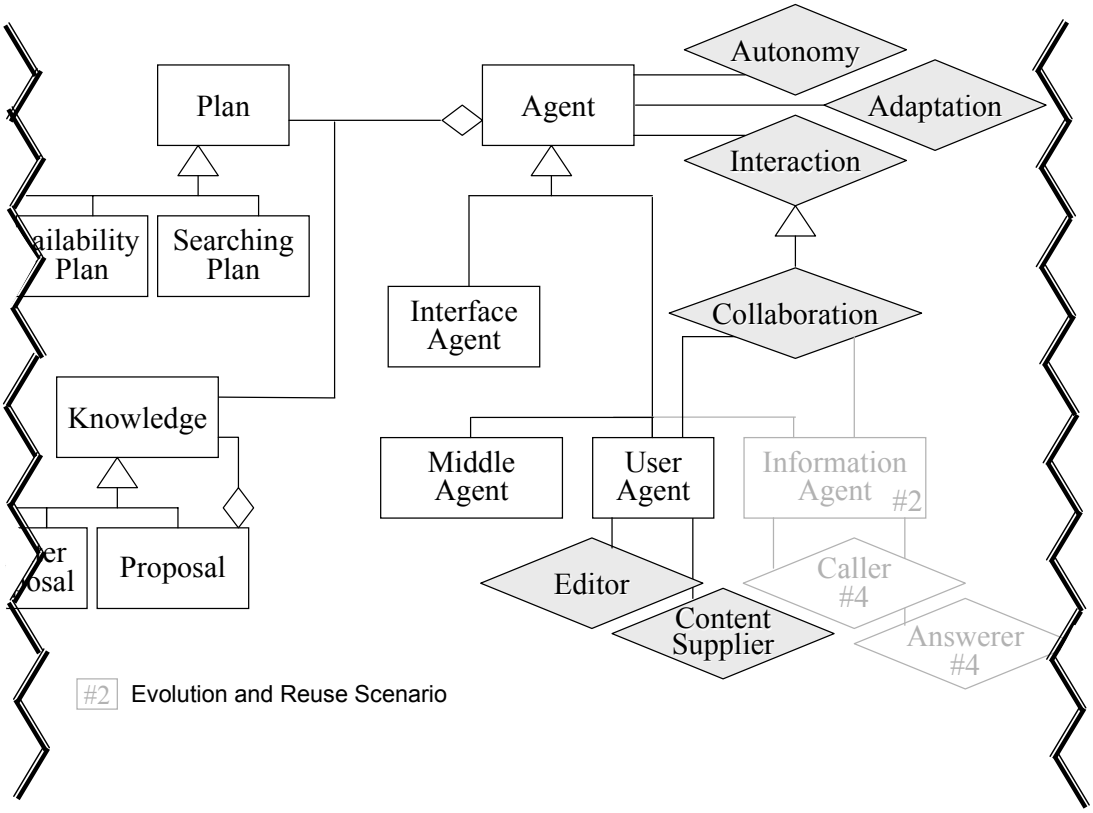


Figure 7. A Slice of the Aspect-Oriented Design of the MAS

The experiment subjects developed two versions of the Portalware system, using both the aspect-oriented and pattern-oriented techniques. Figures 6 and 7 represent respectively slices of the pattern-oriented and aspect-oriented designs for the Portalware MAS. Figure 6 shows a combination of different design patterns to address the MAS concerns. Each pattern is surrounded by a dotted line. In Figure 7, a diamond shape is used to express aspects. Each diamond may be related to one or more rectangles used to describe classes. This relationship is expressed as a line from the aspect to a class. Those figures also illustrate some changes required in the maintenance and reuse scenarios for further clarification in the Section 4.2.

3.5. The Subjects and Study Phases

The study was divided into two major phases: (1) the Construction phase, and (2) the Reuse and Evolution phase. In the Construction phase, three individuals were asked to develop the selected MAS (Section 3.4) using the investigated techniques. The subjects have participated both in the development of the aspect-oriented (AO) system and in the development of the pattern-oriented (PO) system. Among them, 3 were PhD students at PUC-Rio. They were asked to report problems in using the investigated techniques, and associated methods and tools. The set of agent-oriented models, OO design and implementation were based on the same requirement specifications and satisfying the same set of

scenarios. We tried to reduce the possibility that the development of the second system (the pattern-oriented system) benefited from the development of the first system (the aspect-oriented system).

The Reuse and Evolution phase involved the same subjects. The goal of this phase was to compare the reusability and maintainability of the PO solution and the AO solution. To evaluate the modifiability and extendibility of the produced systems, a set of relevant change and reuse scenarios to both original designs and code were made as it will be described in the Sections 4.2 and 5.2. Since our metrics selected (Section 4.1.3) are oriented to number of design components and code lines, we had an additional standardization phase before the data collection. This phase aimed to ensure that the two developed MASs implemented the same functionalities. This phase also removed problems related to different coding styles.

4. The Evaluation Framework

We developed a framework to evaluate the produced systems in terms of our defined goals, questions, and hypotheses (Section 3). The generated framework is composed of a *quality model* and a *set of reuse and evolution scenarios*. A quality model emphasizes and connects the investigated qualities, factors, and criteria [12]. It was also particularly useful for the metric selection and data interpretation. The produced systems are also evaluated through a set of scenarios generated during the case study in an attempt to verify the support degree of the used abstractions for MAS reuse and evolution. Our evaluation framework was validated based on the Kitchenham's measurement framework [23, 24].

4.1. The Quality Model

Quality models are constructed in a tree-like fashion since quality is actually a composite of many other interacting qualities [12]. The notion of software quality is usually captured in a model that depicts other important qualities (which we termed here as factors) and their relationships with primary qualities. Our quality model is composed of 4 different elements: (i) qualities, (ii) factors, (iii) criteria, and (iv) metrics. The qualities are the attributes that we want primarily observe in the software system. The factors are the secondary quality attributes that influence the defined primary qualities. The criteria are related to the well-established software engineering principles that are essential to the achievement of the qualities and their respective factors [12].

Figure 8 presents the elements of our quality model. The upper branches hold important high-level quality factors that we would like to quantify. Each quality factor is composed of lower-level criteria. The criteria are easier to measure than the factors; thus actual measures (metrics) are proposed for the criteria. The definition of our quality model was accomplished in parallel with the use of GQM methodology (Section 3). The following subsections describe the elements of our quality model.

4.1.1. The Qualities

The first obligation of any software-measurement activity is the identification of the primary quality attributes and software artifacts we wish to measure. This study focus on the evaluation of the maintainability and reusability attributes based on distinct artifacts of a MAS, such as its design models and source code.

Reusability and Maintainability. Reusability is the ability of software elements to serve for construction of other different elements in the same software system or across different ones. In this study, we are interested in evaluating the reusability of agency concerns in design and code. Maintenance is the activity of modifying a software system after initial delivery. Software maintainability is the ease with which the software components can be modified. Maintenance activities are classified into four categories [32, 12]: corrective maintenance, perfective maintenance, adaptive maintenance, and evolution. This work focused on MAS evolution, i.e. the addition or elimination of agency concerns.

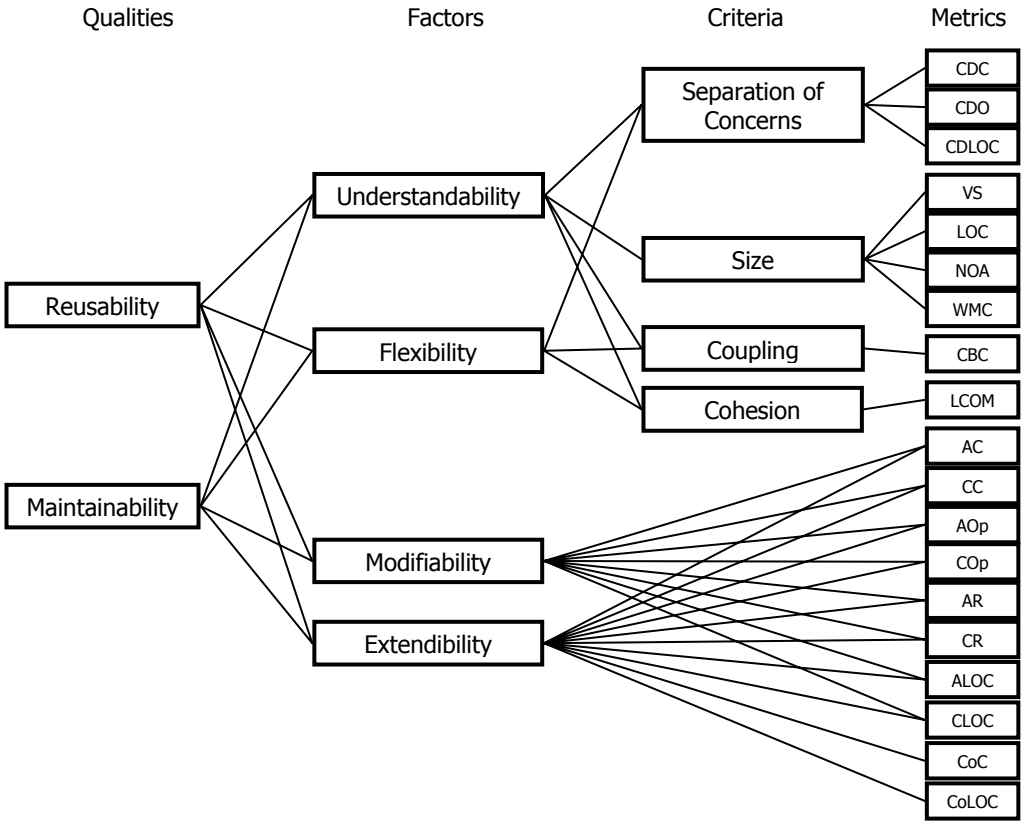


Figure 8. The Quality Model

4.1.2. The Factors and Criteria

The model emphasizes that similar factors are useful for the promotion of maintainability as well as reusability. This similarity is related to the fact the reuse and maintenance activities encompass common cognitive tasks. Flexibility, understandability, modifiability, extendibility are the central factors to promote reuse and maintainability [12, 28, 32]. Both kinds of activities require software abstractions to support understandability and flexibility. An understandable system enhances its own maintainability and reusability because most maintenance and reuse activities involve software engineers in trying first to understand the system components to any further system modification or extension. In addition, a software system needs to be flexible enough to support the addition and removal of functionalities and the

reuse of its components without making a lot of effort. The modifiability and extendibility factors also influence maintainability and reusability and are measured based on the scenarios presented in Section 4.2.

In our model, the understandability factor is composed of the following criteria: (i) size, (ii) coupling, (iii) cohesion and (iv) separation of concerns. The cohesion of a component is a measure of the closeness of the relationship between its components [41]. A component should implement a single logical function. Coupling is an indication of the strength of interconnections between the components in a system. Highly coupled systems have strong interconnections, with program units dependent on each other [41]. Coupling and cohesion affect the understandability because a component of the system can not be understood without reference to the others components to which it is related. The size of design and code may indicate the amount of effort needed for understanding the software components. And the separation of concerns criterion is a predictor of understandability because the more localized the concerns of the system are the easier it is to understand them.

The flexibility factor is influenced by the following criteria: (i) coupling, (ii) cohesion, and (iii) separation of concerns. High cohesion, low coupling and separation of concerns are desired characteristics because it means that a component represents a single part of the system and the system components are independent or almost independent. Further, the systems concerns are not scattered and tangled. If it becomes necessary to add, remove or reuse functionality, it is localized in a single component and the maintenance and reuse activities are flexibly restricted to this isolated component.

4.1.3. The Metrics

The metrics for Size, Coupling, Cohesion and Separation of Concerns were selected and defined to evaluate the techniques and produced systems with respect to the degree of system maintainability and reusability supported. We reused and refined some classical metrics [9, 11, 12] and defined new ones that capture important notions for the context of this study. Although a large body of research in software metrics has been focused on procedural or OO software, there is no software metric for aspect-oriented software until now. So we tailored some object-oriented metrics to apply in aspect-oriented software. In fact, we reworked the definition of the metrics since we should be able to compare OO designs and code with aspect-oriented designs and code. Each metric definition was extended to be applied in a way that is independent from the paradigm, supporting the generation of comparable results. Further, we proposed some metrics of separation of concerns, which were inspired in the Lopes work [27]. Table 2 associates the metrics with the questions generated by the use of GQM and the criteria defined by the quality model. In the following, we present each metric in terms of its definition and its relevance to the system maintainability and reusability.

Metrics	Answered Questions	Criteria
Vocabulary Size (VS)	How many components are there?	Size
Lines of Code (LOC)	How many lines of code are there?	Size
Number of Attributes (NOA)	How many attributes are there?	Size
Weighted Methods per Component (WMC)	How many methods and advices are there?	Size
Coupling Between Components (CBC)	How high is the coupling between components?	Coupling
Lack of Cohesion in Methods (LCOM)	How high is the cohesion of the systems components?	Cohesion
Depth of Inheritance Tree (DIT)	How high is the coupling between components? How high is the cohesion of the systems components?	Coupling and Cohesion
Concern Diffusion over Components (CDC)	How well are the agency concerns localized? (and sub-questions)	Separation of Concerns
Concern Diffusion over Operations (CDO)	How well are the agency concerns localized? (and sub-questions)	Separation of Concerns
Concern Diffusions over LOC (CDLOC)	How well are the agency concerns localized? (and sub-questions)	Separation of Concerns

Table 2. Association between the Metrics, the GQM Questions and the Quality Model Criteria

a) Vocabulary Size (VS)

Definition: VS counts the number of system components, i.e. the number of classes and aspects into the system. This metric measures the system vocabulary size. Each component name is counted as part of the system vocabulary. The component instances are not counted.

Relevance: The higher the vocabulary size, the more difficult it is to understand the system. The more difficult it is to understand the system, the harder it is to find the components that must be changed during evolution activities or the components that provide the required functionalities during reuse activities.

b) Lines of Code (LOC)

Definition: It counts the number of code lines. This is the traditional measure of size. Documentation and implementation comments as well as blank lines are not interpreted as code. Different programming styles usually bias the results of this metric application. In our study, we overcame this problem by ensuring the same programming style was used in both projects.

Relevance: The higher the number of code lines, the more difficult it is to understand the system. The higher the number of code lines, the harder it is to find the lines that must be changed during evolution activities or understand the implementation of the required functionalities during reuse activities.

c) Number of Attributes (NOA)

Definition: This metric counts the internal vocabulary of each component, i.e. the number of attributes of each class or aspect. Inherited attributes are not included in the count.

Relevance: The higher the number of attributes per component, the more difficult it is to understand the system. The higher the number of attributes, the harder it is to find the locals that must be changed during

evolution activities or understand the implementation of the required functionalities during reuse activities.

d) Weighted Methods per Component (WMC)

Definition: This metric measures the complexity of a component in terms of its operations. Consider a component C_1 with operations (methods or advices) O_1 ,..., O_n. Let c_1 ,…, c_n be the complexity of the operations. Then: WMC = c_1 + ... + c_n. This metric originally does not specify the operation complexity measure, which it should be tailored to the specific contexts. In this study, the method complexity measure is get by counting the number of parameters of the operation, assuming that a operation with more parameters than another is likely to be more complex.

Relevance: The higher the number and complexity of operations per component, the more difficult it is to understand the system. The higher the number and complexity of operations per component, the harder it is to find the locals that must be changed during evolution activities or understand the implementation of the required functionalities during reuse activities.

e) Coupling Between Components (CBC)

Definition: This metric is defined for a class or aspect as a count of the number of other classes and aspects to which it is coupled. It counts the number of classes that are used in attribute declarations, formal parameters, return types, throws declarations and local variables, and classes and aspects from which attribute and method selections are made. An aspect is also said to be coupled to a component if it has a pointcut that define a join point where it cut across another aspect or class. Even a component A accesses a component B twice or more, we will count just one time.

Relevance: The component understanding involves the understanding of the components to which it is coupled. So the larger the number of couples of a component, the more difficult it is to understand the system. In order to improve modularity and promote encapsulation, inter-component couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Excessive coupling between components is detrimental to modular design and prevents reuse. The more independent a component is, the easier it is to reuse it in another application.

f) Lack of Cohesion in Methods (LCOM)

Definition: This metric measures the cohesion of a component. If a component C1 has *n* operations (methods and advices) O_1,…, O_n then $\{I_j\}$ is the set of instance variables used by operation O_j. Let |P| be the number of null intersections between instance variables sets. Let |Q| be the number of non-empty intersections between instance variables sets. Then:

LCOM = |P| – |Q|, if |P| > |Q|
LCOM = 0 otherwise

LCOM measures the amount of method/advice pairs which do not access the same instance variable. As such it is a measure of lack of cohesion.

Relevance: The higher the degree to which different actions performed by a component contribute towards distinct functions, the harder it is to reuse and maintain the component or one of its functionalities.

g) Depth of Inheritance Tree (DIT)

Definition: DIT is defined as the maximum length from the node to the root of the tree. It counts how far down the inheritance hierarchy a class or aspect is declared.

Relevance: The deeper a class or aspect is in the hierarchy, the greater the number of methods, advices and attributes it is likely to inherit, making it more difficult to understand it. Components which inherit attributes and operations are coupled to their super-components. Changes to the super-components must be made carefully as the changes propagate to all components which inherit their characteristics. If a component inherits attributes and operations from a super-component, the cohesion of that component is reduced. The deeper a component is in the inheritance tree, the harder it is to reuse it because all of its super-components should be understood.

h) Concern Diffusion over Components (CDC)

Definition: CDC counts the number of components whose sole purpose is to assist in the implementation of a concern. Furthermore it counts the number of components that access the components whose sole purpose is to assist in the implementation of the concern, i.e. use them in attribute declarations, formal parameters, return types, throws declarations and local variables, or call their methods.

Relevance: This metric measure the degree to which a single concern in the system maps to the components in the software design. The more direct a concern maps to the components, the easier it is to understand it. The more direct a concern maps to the components, the fewer components will be changed during maintainability activities or the fewer components should be understand and extend during reuse activities.

i) Concern Diffusion over Operations (CDO)

Definition: CDO counts the number of operations whose sole purpose is to assist in the implementation of a concern. Furthermore it counts the number of methods and advices that access any component whose sole purpose is to assist in the implementation of the concern, i.e. use them in formal parameters, return types, throws declarations and local variables, or call their methods. Constructors also count as methods.

Relevance: One way of measuring the code tangling is by counting the number of operations affected by concern code. The more operations the concern affect, the harder it is to understand it. The more operations the concern affect, the more scattered is the concern and, therefore, it is more difficult to reuse it and more operations will be changed during maintainability activities.

j) Concern Diffusions over LOC (CDLOC)

Definition: In order to capture the tangling of concern code within implementation, the following metric can be defined:

CDLOC = number of transition points between concerns

We have to shadow the parts of the code that deal with the assessed concern. Transition points are the points in the code where there is a transition from a non-shadowed area to a shadowed area and vice-versa. The intuition behind it is that they are points in the program text where there is a "concern switch". For each implementation, the programs texts are analyzed line by line in order to count transition points. The higher CDLOC, the more intermingled the concern code is within the implementation of the components; the lower CDLOC, the more localized the concern code is.

In our case study, the identification of the concern code follows the guidelines:

a) Classes whose sole purpose is to assist in the implementation of each agency concern are treated in a special way: both the declaration and its methods are shadowed as a single block. Method invocations to instances of those classes are shadowed.
b) Aspects whose sole purpose is to assist in the implementation of each agency concern are treated in a special way: both the declaration and its methods and advices are shadowed as a single block. Method invocations to instances of those aspects are shadowed.
c) Methods whose sole purpose is the implementation of an agency concern are shadowed; calls to those methods are also shadowed. Note that these methods are not part of the classes or aspects whose sole purpose is to implement an agency concern.
d) Variable declarations used for holding agency concerns' state (e.g. roles, agent type, autonomy…) are also shadowed, as a block; the use of these variables is also shadowed.
e) Method signatures whose parameters contain reference to objects that implement an agency concern are shadowed. Note that these methods are not part of the definition of that concern.
f) Aspect declarations that include references to other aspects (encapsulating other concerns) are shadowed. This is the case of the "dominates" statement.
g) The application of guidelines a-f can generate two or more shadowed blocks for the same concern, which are in a sequence; then this set of blocks should be unified as a single block.
h) If two blocks of the same concern are not in sequence but could be we should count then as a single block.

CDLOC results are relative to the concerns that were being searched and the method for counting transition points. Any small variation of these two factors results in drastic changes of the numbers. Figure 9 shows a sample of shadowing of the Interaction concern in the PO project.

Relevance: The more scattered and tangled a concern is, the more difficult it is to understand it, the more points of the code will be affected during evolution activities and the harder it will be to reuse it.

```
public class PAgent {

    private String agentName;
   ...
   ...
    protected Interaction theInteraction;
    protected Autonomy    theAutonomy;
    protected Adaptation  theAdaptation;

//  Constructors

    public PAgent(String aName, Vector pl) {
        init();
       agentName = aName;
        theInteraction = new Interaction(this);
        theAutonomy = new Autonomy(this);
        theAdaptation = new Adaptation(this);
        planList = pl;
        System.out.println(" Name == " + agentName);
   }
   ...
   ...
   ...
/* Interface  for  Interaction */
   public void receiveMsg(Message msg)
   {
       theAutonomy.makeDecision(msg);
       theAdaptation.adaptBeliefs(msg);
   }

   public void outcomingMsg(Message msg)
   {
       theInteraction.outcomingMsg(msg);
   }
}
```

Figure 9. An example of code shadowing

4.2. The Reuse and Evolution Scenarios

We have simulated simple and complex changes related to agency concerns to both the PO and AO solutions in order to measure their modifiability and extendibility support. We have selected 7 change and reuse scenarios which are recurrent in large-scale MAS, such as inclusion of new agents, reuse of roles and collaborative capabilities, and so forth. The list of the scenarios is the following:

S1) Change on the Agent Roles (Evolution)
S2) Creation of an Agent Type (Evolution)
S3) Reuse of the Agenthood Concern (Reuse)
S4) Inclusion of Collaboration in an Agent Type (Reuse and Evolution)
S5) Reuse of Roles (Reuse)
S6) Creation of a new Agent Instance (Evolution)
S7) Change of the Agenthood Definition (Reuse and Evolution)

For each concern change made to the system, the difficulty of the concern modifiability is defined as the sum of following items: (1) number of components (aspects/classes) added, (2) number of components changed, (3) number of relationships included, (4) number of relationships changed, (5) number of new

LOCs, (6) number of modified LOCs, (7) number of operations (methods/advices) added, (8) number of operations (methods/advices) changed.

For each attempt made to reuse some concern, the difficulty of extendibility is defined as the sum of modifiability items and the following items: (9) number of copied entities, and (10) number of copied LOCs. All these items were observed from the structural and behavioral design models and code. Those elements in aspect-oriented development and those in pattern-oriented development are comparable because they represent the same concerns of high-level agent models. We describe below the new requirements and associated maintenance and reuse tasks needed to satisfy these requirements. Section 5.2 and Table 3 overviews the main results of each reuse and maintenance scenarios and highlights the main differences detected between the investigated solutions.

5. Results

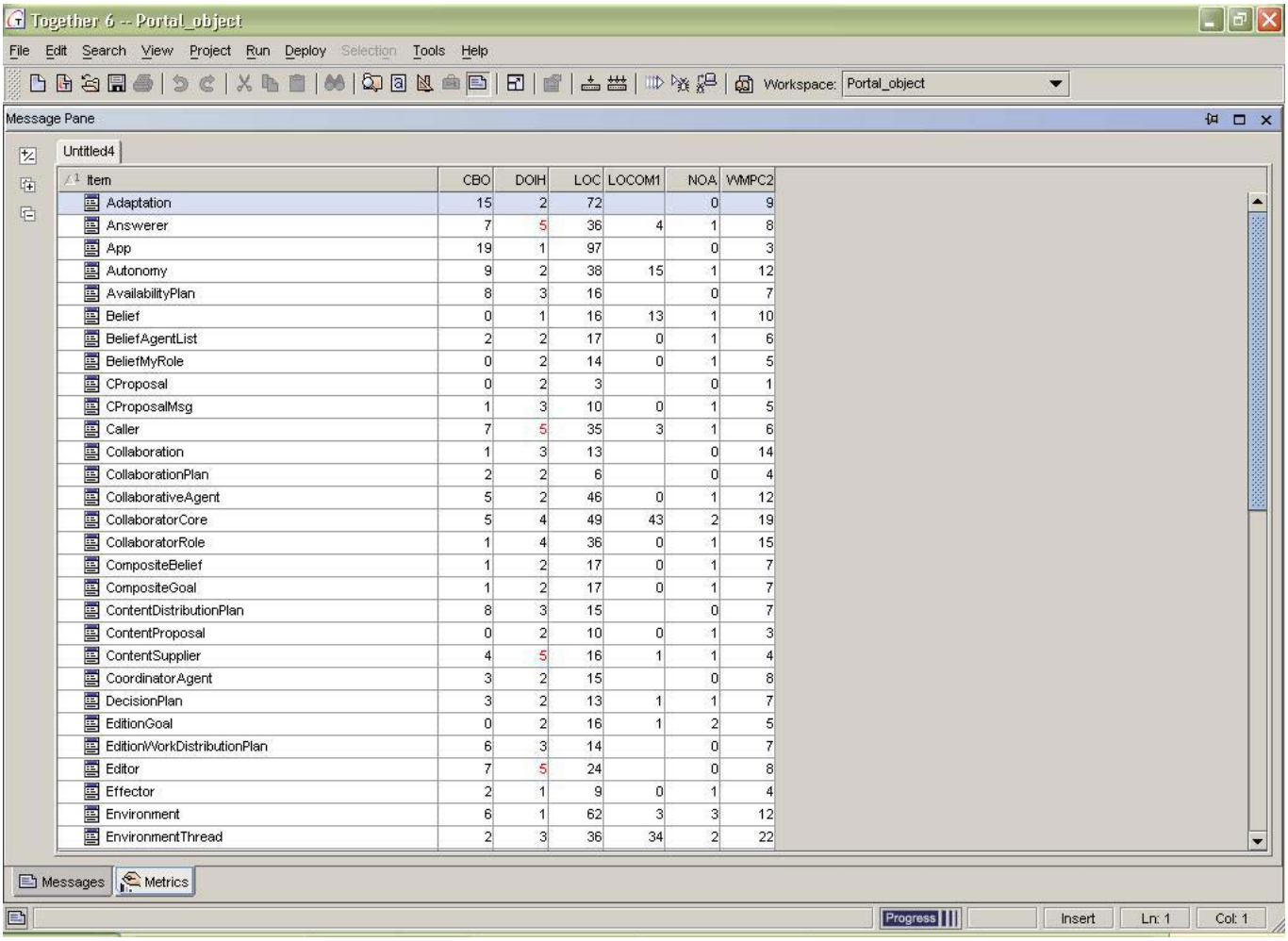
This section presents the results of the measurement process. Section 5.1 overviews the results obtained at the end of the MAS construction phase. The data were collected based on the set of defined metrics (Section 4.1.3). All data gathered in the construction phase is available in the Appendices B. Section 5.2 describes the results of the MAS evolution and reuse phase based on the selected scenarios and associated metrics (Section 4.2). The discussion about the results is concentrated at the Section 6.

5.1. The MAS Construction Phase

The data was partially gathered by the CASE tool Together 6.0. It supports some metrics: LOC, NOA, WMC (called WMPC2 in Together), CBC (called CBO in Together), LCOM (called LOCOM1 in Together) and DIT (called DOIH in Together). Figure 10 shows a sample of the results of some classes of the PO system based on Together. The following subsections present the results of the size metrics (Section 5.1.1), the coupling and cohesion metrics (Section 5.1.2), and the separation of concerns metrics (Section 5.1.3).

5.1.1. Results of the Size Metrics

VS. The *external vocabulary* (i.e. the number of system components) of the aspect-oriented MAS is simpler than in the object-oriented MAS, since the amount of design and implementation components in the later (VS = 60) was higher than in the former (VS = 56). The main reason for this result is that the *Role* and *Mediator* patterns required additional classes to address the decomposition and composition of multiple agent roles and behavior properties, respectively. The aspect-oriented solution does not need such additional classes since the composition is specified by the pointcuts which are defined internally to the aspects.



Item	CBO	DOIH	LOC	LOCOM1	NOA	WMPC2
Adaptation	15	2	72		0	9
Answerer	7	5	36	4	1	8
App	19	1	97		0	3
Autonomy	9	2	38	15	1	12
AvailabilityPlan	8	3	16		0	7
Belief	0	1	16	13	1	10
BeliefAgentList	2	2	17	0	1	6
BeliefMyRole	0	2	14	0	1	5
CProposal	0	2	3		0	1
CProposalMsg	1	3	10	0	1	5
Caller	7	5	35	3	1	6
Collaboration	1	3	13		0	14
CollaborationPlan	2	2	6		0	4
CollaborativeAgent	5	2	46	0	1	12
CollaboratorCore	5	4	49	43	2	19
CollaboratorRole	1	4	36	0	1	15
CompositeBelief	1	2	17	0	1	7
CompositeGoal	1	2	17	0	1	7
ContentDistributionPlan	8	3	15		0	7
ContentProposal	0	2	10	0	1	3
ContentSupplier	4	5	16	1	1	4
CoordinatorAgent	3	2	15		0	8
DecisionPlan	3	2	13	1	1	7
EditionGoal	0	2	16	1	2	5
EditionWorkDistributionPlan	6	3	14		0	7
Editor	7	5	24		0	8
Effector	2	1	9	0	1	4
Environment	6	1	62	3	3	12
EnvironmentThread	2	3	36	34	2	22

Figure 10. Using Together to Apply the Defined Metrics

NOA. The *internal vocabulary* (i.e. the number of attributes) of the PO solution components is more complex than in the AO components. The number of attributes of the MAS components in the PO solution was higher than in the AO solution. This problem happened because the agent objects in the PO project need to have explicit references to the objects representing the three agent basic properties and the agent roles. For example, the Agent class (the class that encapsulates the agenthood structure and behavior), is composed of 9 attributes in the PO system, while it has 6 attributes in the AO system. The Plan class and their specific subclasses have attributes that hold pointers to the Role classes, since a number of roles are performed during a plan execution. In the OA project, the plan classes and role aspects are bound implicitly which reduces the internal complexity of these classes. In addition, the collaboration concern is modularized directly by the Collaboration and Role aspects, while it is spread over 5 different classes (Collaboration, CollaborationCore, CollaborationRole, CollaborativeAgent, and Role) in the PO project and, as a consequence, this is an increasing factor in the NOA since each of these classes need to have references to each other.

LOC. The LOC was 1445 in the PO implementation and 1271 in the AO implementation, i.e. the PO code has 174 lines more than the AO code. In general, the implementation of each (plan, role) couple in the PO code was 10 lines higher than in the AO code. It is because in the PO solution there is a need for

additional method calls on the Role subclasses to: (i) activate and deactivate roles dynamically, and (ii) to get references to the role objects. These calls are not needed in the aspect-oriented project because the composition is specified by the pointcuts in the role aspects. In addition, less code lines were needed to implement the collaboration concern in the AO system since the PO system implements additional classes of the Role pattern, as described previously. Regarding the agenthood concern, its main component, the Agent class, has 118 lines in the PO implementation and 95 lines in the AO implementation. This difference occurs because this class needs to manage explicitly the basic agent properties in the PO solution. The interaction concern is the only one that less code lines were needed to be implemented in the PO project; the AO project uses 59 lines, while the PO project uses 54 lines.

WMC. In general, the measures here pointed out higher WMCs in the AO solution. It happens because the modularization of some concerns in the aspects requires the context to be recaptured by means of pointcuts. Some examples of the WMC measures: (i) it is 9 for the Adaptation class in the PO solution, while it is 10 in the AO solution, (ii) it is 12 for the Autonomy class in the PO solution, while it is 14 in the AO solution, and (iii) it is 17 for the Interaction concern in the PO solution, while it is 25 in the AO solution. The recurrent situation to these examples is that the Interaction, Adaptation and Autonomy aspects receive the agent object reference as parameters in the pointcuts, while such reference is accessed in a local attribute in the respective classes of the PO project. In the particular of the Interaction concern, there is an additional conclusion: the bigger difference than in the other cases occurs because the Interaction aspect is able to encapsulate more functionality than the Interaction class in the PO project, where such concern is more spread over distinct agent classes. This leads a higher number of advices/methods in the AO project and, as a result, a higher WMC. Since the Agent class in the PO solution have methods related to the Interaction concern, such as *receiveMsg(Message msg)* and *sendMsg(Message msg)*, its WMC is higher than the WMC of the Agent class in the AO solution (33 vs. 29). The WMC of the collaboration concern is higher in the PO system since it crosscuts different classes. The WMC of the Collaboration aspect is 12, while it is 14 in the Collaboration class, even we do not consider the WMCs of the additional classes of the role pattern: CollaborativeAgent (WMC = 12), CollaborationCore (WMC = 19), CollaborationRole (WMC = 15). Finally, the WMCs of role classes is lower than WMCs of role aspects: answerer (8 vs. 9), caller (6 vs. 12), content supplier (4 vs. 7), and editor (8 vs. 15).

5.1.2. Results of the Coupling and Cohesion Metrics

DIT. There are major problems in the PO design when considering this metric. The use of the role pattern lead to a 5-level hierarchy to structure the agent roles, which potentially conducts to a high inheritance coupling [41]. In the AO solution, the DIT = 1 since roles are encapsulated in aspects which crosscuts the agent type hierarchy. Moreover the use of the mediator pattern lead to a 3-level hierarchy to structure the agent types, while the same hierarchy in the AO system has 2 levels. This problem occurs since it is needed to create an additional level (the CollaborativeAgent class) in the PO solution in order to separate the collaborative agents from the non-collaborative ones. In the AO solution, this separation is accomplished transparently by the collaboration aspect that defines which agent types are collaborative and, as a consequence, it does not affect the size of the agent type hierarchy. The use of the Mediator pattern also determines a second problem in the DIT measures of the PO solution. Since it establishes one class should mediate the interaction (the Property class) between the classes representing the 3 basic properties, this solution results in a DIT = 2. This mediation class is not required in the AO system since the property interaction is handled directly in the respective aspects representing the basic properties.

CBC. There is a significant difference between the CBC of the two solutions for the Agent class. The CBC is 12 in the PO system and 9 in the AO solution. This happens since the Agent class needs to have explicit references to the classes representing the basic agent properties. The reverse is also necessary, i.e. the classes Autonomy, Interaction, and Adaptation requires explicit references to the agent object. In the AO project, only the aspects have reference to the Agent class. The coupling is higher also for the role concern in the PO solution. This is provoked by the fact that the roles need to access the methods on the several classes of the role pattern, as described before. The aspect-based solution alleviates this problem since the role implementation is centralized in an aspect. Here are two examples: (i) the CBC for the editor role is 23 in the PO solution, while it is 18 in the AO solution, and (ii) the CBC for the caller role is 25 in the PO solution, while it is 19 in the AO solution.

LCOM. A low LCOM value indicates high coupling between methods (i.e. high cohesion), which however indicates high testing effort because many methods can affect the same attributes. Most components of the PO system produced better results in terms of cohesion than the components of the AO system. For example, the LCOM for the Agent class is 50 in the PO project and 57 in the AO project. The LCOM for the Interaction concern is 29 in the PO project and the 48 in the AO project. The PO system has also produced lower values to the roles: the Answerer role (4 vs. 6), the Caller role (3 vs. 10), the Content Supplier role (1 vs. 3).

5.1.3. Results of the Separation of Concerns Metrics

CDC. Every MAS concern required more components in the definition of the PO solution than in the AO solution. All roles required more than 5 classes to their definition, while one single aspect is able to encapsulate each system role: Answerer (6 vs. 1), Caller (7 vs. 1), ContentSupplier (6 vs. 1), and Editor (7 vs. 1). The agency properties also need more components in the PO design and implementation: adaptation (3 vs. 1), autonomy (3 vs. 2), collaboration (15 vs. 6), and interaction (7 vs. 6). Finally, more components are also used in the PO design and implementation of agent types -- 37 vs. 33 both for information and user agent types -- and their respective instances – 3 vs. 1 for information agents and 4 vs. 2 for user agents.

CDO. Again all concerns require more operations (methods/advices) in the PO system than in the AO system. Most concerns in the PO solution are implemented with more than the double number of operations used in the AO system. For instance the case for the Adaptation concern and the Autonomy concern. There are some cases where the difference is even bigger (for example, all roles), less than the double (for example, collaboration and both agent types), or almost the same (for example, agenthood and Interaction).

CDLOC. The measures here also pointed out that the AO solution was more effective to modularize the MAS concerns. The results for the agent types did not present any difference. All other cases, except the agenthood concern, were better encapsulated in the AO design and implementation. The detected differences were very significant in the following concerns: (i) the basic agent properties, (ii) the agent roles, (iii) the agent instances, and (iv) the collaboration property. The problem with the agenthood concern happened since the Interaction aspect specifies in its definition that it must to be executed before the Collaboration aspects, increasing the number of transition points.

5.2. The MAS Evolution and Reuse Phase

In the following, we discussed each scenario and its results in both pattern-oriented and aspect-oriented solutions. Table 3 overviews the main results of the evolution and reuse scenarios. Figures 6 and 7 illustrate some basic changes in the design artifacts, which were necessary in the scenarios S2 and S4.

Scenario S1 – Change on the Agent Roles

To improve the quality of the produced portal material, we decided to include the reviewer role into the system in order to examine each of the content segments provided by the content suppliers and change such segments if this is thought desirable or necessary. To reflect this new system requirement, new agents were included into the system to represent users which will play the reviewer role. So a new role and respective plans were incorporated to the system in order to encapsulate the reviewer behavior and associated plans. This role must be associated with the UserAgent type that is the agent type that models the system users. This change resulted in similar impact to both PO and AO solutions. However, some additional lines were necessary in the change of the PO system.

Scenario S2 – Creation of an Agent Type

Portalware users often need to search for information which is stored in multiple databases and available on the web in order to produce the material required by the editors. As a consequence, a new agent type, called InformationAgent, was included into the system for automating this time-consuming task. Each InformationAgent instance contains different searching plans and is attached to an information source. This change resulted in similar impact to both PO and AO solutions. However, this change scenario triggered two different change scenarios (S3 and S4) which are described immediately below.

Scenario S3 – Reuse of the Agenthood Concern

Since all system agent types incorporate the agenthood features, the incorporation of the InformationAgent type required the reuse of these features. As presented in Table 3, this scenario also resulted in similar changes to both PO and AO systems.

Scenario S4 – Inclusion of Collaboration in an Agent Type

The inclusion of the information agent also required the reuse of the collaboration concern defined in the system previously. It is because information agents collaborate with each other when an information agent is not able to find the required information in the respective information source. As a result, information agents play the caller and answerer roles respectively to: (1) call another information agents and ask for an information, and (2) receive requests and send the search result to the caller. As a result, this scenario involved two main tasks: (1) the reuse of the predefined collaboration concern in the context of the InformationAgent, and (2) creation and attachment of 2 roles to the InformationAgent type. This was the scenario that resulted in major differences between the changes in the PO solution and the AO solution: (1) the PO code required 20 lines more than the AO code, (2) more relationships were added in the PO design, and (3) eight lines were removed from the AO code while no line was changed in the PO code.

	MODIFIABILITY																			
	EXTENDIBILITY																			
	Changed Entities		Changed Operations		Added Entities		Added Operations		Changed Relations.		Added Relations.		Added LOCs		Changed LOCs		Copied Entities		Copied LOCs	
	PO	**AO**	**PO**	**AO**	**PO**	**AO**	**PO**	**AO**	**PO**	**AO**	**PO**	**AO**	**PO**	**AO**	**PO**	**AO**	**PO**	**AO**	**PO**	**AO**
S1	1	1	3	3	5	5	2	3	0	0	15	15	101	98	1	1	-	-	-	-
S2	0	0	2	2	4	4	0	0	0	0	10	10	84	86	0	0	-	-	-	-
S3	0	0	2	2	4	4	0	0	0	0	10	10	84	86	0	0	0	0	0	0
S4	0	0	2	3	8	8	0	0	0	0	29	25	188	167	0	8	0	0	0	0
S5	1	1	2	1	0	0	1	1	0	0	4	2	16	14	0	0	0	0	6	6
S6	0	0	0	0	0	0	0	0	0	0	0	0	15	15	0	0	-	-	-	-
S7	5	1	0	0	0	0	0	0	5	2	1	1	0	0	5	1	0	0	40	0

Table 3. The Results of the Reuse and Maintenance Scenarios

Scenario S5 – Reuse of Roles

The introduction of information agents into the system made information services available to the other agents. Since some user agents play the content supplier role, it could automate the task of selecting information relevant to certain contexts in behalf of their respective users. In this sense, the user agents should use the services of the information agents playing a caller role, which is already defined in the MAS. In other words, the caller role needs to be reused and attached to UserAgent type and the ContentSupplier role in both AO and PO solutions. This reuse scenario required more effort in the PO project than in AO project in the following items: (1) number of changed operations (2 vs. 1), (2) number of added relationships (4 vs. 2), and (3) number of added LOCs (16 vs. 14).

Scenario S6 – Creation of a new Agent Instance

This scenario investigated the impact of adding new agent instances into the system. In particular, we created a new instance of the UserAgent type to play the content supplier role. This maintenance scenario required the same changes in the PO and AO versions of the MAS. The addition of 15 LOCs was implemented in the two versions.

Scenario S7 – Change of the Agenthood Definition

The last scenario was created to simulate a really pervasive change in both solutions. With the inclusion of the information agent into the system, all other agent types are able to use its services to achieve their specific goals. So every agent type should be collaborative which implies in the extension of the agenthood definition to include the collaboration concern. However, this scenario requires the reuse of the previous agenthood definition. The AO solution provided better modifiability and extendibility support in this case: (1) 5 components were changed in the PO design and just one in the AO design, (2) 5 relationships were changed in the PO design while 2 ones were modified in the AO design, (3) 40 lines were copied in the PO code and none line in the AO code.

6. Discussion

Although the conclusions can not be extrapolated to all MASs, this study was conducted in a system that includes the canonical features of reactive MASs. The application of the two different OO techniques was conducted by different developers. As stated in Section 3, a number of procedures were considered in order to minimize common problems in empirical software engineering. In addition, the present work provides an experimental framework which other MASs developers can use and refine in their specific domains and MAS applications in order to refine the knowledge of the software engineering community about the interplay between agents and objects, and the associated difficulties in designing and implementing MAS with object-oriented techniques. Also we are currently developing a similar empirical study in an open electronic marketplace system, which involves a huge number of types and instances of mobile and cognitive agents. The sections below detail the results and the analysis for each of the stated hypothesis.

6.1. Techniques Comparison (Hypothesis 1)

In general, the results (Section 5) have shown that the AO technique provided better maintainability and reusability support for the selected project (Section 3.4) than the pattern technique. According the metrics application, we can present the following conclusions:

 The aspect-oriented technique produced a more concise MAS, in terms of code lines, external vocabulary and internal vocabulary of the components (Section 5.1.1). The use of design patterns leads to an increase in the number of classes, which are dedicated to encompass limitations of the composition mechanisms of OO programming languages. This conclusion is supported by all size metrics, except the WMC metric.

 The aspect-oriented technique produced more complex operations, i.e advices, than the object-oriented technique (Section 5.1.1). It happens because the modularization of some concerns in the aspects requires the object contexts to be recaptured by means of pointcuts. This result is supported by the WMC values.

 The pattern-oriented technique leads to the abuse of the inheritance mechanism, which is fundamental to establish high inheritance couplings. This problem was detected by the DIT values (Section 5.1.2).

 The pattern-oriented technique produced components more highly coupled than the aspect-oriented technique. This is a consequence of the lack of expressive power in this technique to modularize MAS concerns. This result is supported by the CBC metric (Section 5.1.2).

 The pattern-oriented technique produced better results in terms of cohesion than the aspect-oriented technique (Section 5.1.2). The lack of cohesion in the aspects occurs because an aspect is aimed to encapsulate behaviors which act over different components. However, these behaviors cannot be directly related to each other, producing high LCOM values (i.e. low cohesion).

 The aspect-oriented technique clearly provides better support for separation for MAS concerns. It was found the aspect-oriented mechanisms provide improved support to modularize agency concerns. This finding is supported by all separation of concerns metrics (Section 5.1.3).

 In terms of the evolution and reuse scenarios (Section 5.2), the aspect-oriented technique presented better results. This conclusion is supported by the following metrics: changed entities, changed relationships, added relationships, added LOCs, and copied LOCs.

However, we need empirical evidence to support the hypothesis that the AO technique produces MASs which are easy to understand in terms of spent time. It is because we did not measure the effort of using the investigated techniques in terms of understanding time. This is the focus of our future work. In this sense, we plan use some metrics such as Time spent on Understanding (TU), Understanding Rate (UR), and Time to Change (TC) [36] to investigate this aspect. TU is defined as the time spent on understanding the produced design or code. UR is defined as the understanding rate (1-5) of the produced design or code. TC is defined as the time spent to understand which components should be reused or changed and to make modification to the software artifacts.

6.2. Agents x Objects (Hypothesis 2)

Our hypotheses stated that MAS design and implementation could not be easily supported by the object-oriented abstractions. This hypothesis was based on the assumption that the structure and behavior of complex agents is very complex since it include multiple concerns. However the collected results did not support this hypothesis, since we found advanced OO techniques, supported by effective methods, may deal successfully with concerns of reactive MASs. The additional abstractions (patterns and aspects) were important to cover conceptual gaps between agents and objects, as discussed in the following.

Agent types, classes and inheritance. The pure OO abstractions were useful to encapsulate some basic agency concerns. Classes and the inheritance mechanism provided direct support for structuring the multiple agent types. Inheritance was interesting to promote reuse of operations common to all agent types, such as belief and plan updates.

Patterns and aspects minimized the misalignments. The main difference between the aspect-oriented method and the pattern-oriented method is that the former explored aspects to separate roles and agent properties (such as adaptation, interaction, autonomy, collaboration, and so on), while the later uses known patterns. This separation was very helpful to promote traceability among agent-oriented models and OO designs and code. However, the aspect notion provided better traceability since when an agent property or role was added or removed in the agent-oriented modeling, it was more directly done in the aspect-oriented design and code (Section 5.2). The aspect abstraction was more appropriate to deal with roles from the reusability and maintainability viewpoint (Section 5.2). This finding is similar to the ones reported in [40].

The aspect-oriented technique supported better reuse and maintenance. Design patterns have no first-class representation at the implementation level. The implementation of a design pattern, therefore, cannot be reused and, although its design is reused, the MAS developer is forced to implement the pattern many times. Unlike patterns, aspects provide first-class representation at the implementation-level for agency concerns (such as interaction, collaboration, roles, and so on), supporting reuse both at the design and implementation levels. For example, the reuse of the *Collaboration* property in the context of user agents required the association of the Collaboration aspect to UserAgent class, depicting the join points of interest, while in the pattern-based approach, some additional modifications were required to introduce the association as well as the explicit calls to methods defined in the interface of the Collaboration class.

7. Conclusions and Ongoing Work

The separation of MAS concerns is essential to MAS engineers since they can decide to extended and modify such concerns as the system evolves [16]. The development of complex MASs undergo a transition from agent-oriented models, constructed according an existing agent-oriented methodology and their related abstractions, into object-oriented designs and implementation. This transition needs to ensure that the MAS concerns encapsulated by abstractions in high-level agent-oriented models, are successfully mapped to abstractions available in object-oriented programming languages and implementation frameworks. Among the problems inherent in such transition, none is more serious than the difficulty to handle the conceptual differences between agents and objects, requiring the application of well-established principles and their supporting techniques and methods. More generally speaking, there is a need for understanding the relationships between the object-oriented and agent-based paradigms.

In a previous work [33], we have identified several commonalties and differences between agent and object abstractions for the conceptual modeling. However, we have not identified how agent abstractions are related with OO abstractions that have succeeded to design and implement high-quality systems. So this paper presented an empirical study that we organized to compare the maintainability and reusability support of two emerging OO techniques for MAS development. The results have shown that the aspect-oriented technique allowed the construction of a reactive MAS with improved separation of concerns, lower coupling between its components (although less cohesive), more concise, more reusable, and maintainable. Another important conclusion of this empirical study is that the aspect-oriented approach also supported a better alignment with higher-level abstractions from agent-oriented models. Since this is a first exploratory study, to further confirm the findings, other rigorous and controlled experiments are needed. We are currently developing a similar empirical study in an open electronic marketplace system, which involves a huge number of types and instances of mobile and cognitive agents.

References

[1] ACL, 1997. Agent Communication language, FIPA 97 specification part 2, Technical Report, October.

[2] Basili, V., Selby R., Hutchins D. “Experimentation in Software Engineering”. IEEE Transactions on Software Engineering, SE-12, 1986 p. 733-743.

[3] Basili, V., Caldiera, G., Rombach, H. “The Goal Question Metric Approach”. Encyclopedia of Software Engineering - 2 Volume Set, pp 528-532, John Wiley & Sons, Inc. 1994

[4] F. Bellifemine, A. Poggi & G. Rimassi, "JADE: A FIPA-Compliant agent framework", Proc. Practical Applications of Intelligent Agents and Multi-Agents, April 1999, pg 97-108 (See http://sharon.cselt.it/projects/jade for latest information)

[5] Booch, G., Rumbaugh, J., Jaconbson, I.: “The Unified Modeling Language User Guide”. Addison Wesley, 1999.

[6] Bradshaw, J, “Software Agents”. American Association for Artificial Intelligence/ MIT Press, 1997.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. “Pattern-Oriented Software Architecture: A System of Patterns”, John Wiley & Sons, 1996.

[8] Chavez, C., Lucena, C. “Design Support for Aspect-oriented Software Development”. Doctoral Symposium at OOPSLA'2001 and Poster Session at OOPSLA'2001, Tampa Bay, Florida, USA, October 14 – 18, 2001.

[9] Chidamber, S.; Kemerer, C. “A Metrics Suite for Object Oriented Design”. IEEE Transactions on Software Engineering vol. 20 no. 6 June 1994 p476-493.

[10] Elammari, M. and Lalonde, W. “An Agent-Oriented Methodology: High-Level and Intermediate Models”. Proceedings of AOIS 1999 (Agent-Oriented Information Systems), Heidelberg (Germany), June 1999.

[11] Fenton, N. “Softwre Metrics: A Rigorous Approach.”, London: Chapman & Hall, 1991.

[12] Fenton, N and Pfleeger, S. “Softwre Metrics: A Rigorous and Practical Approach.” , 2. ed. London: PWS, 1997.

[13] T. Finin, R. Fritzson, D. McKay, and R. McEntire, "KQML as an Agent Communication Language", Proceedings of the Third International Conference on Information and Knowledge Management, ACM Press, 1994, pp. 456-463.

[14] Gamma, E. et al. “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley, Reading, 1995.

[15] Garcia, A., Silva, V., Chavez, C., Lucena, C. “Engineering Multi-Agent Systems with Aspects and Patterns”. Journal of the Brazilian Computer Society, November, 2002.

[16] Garcia, A., Lucena, C. Software Engineering for Large-Scale Multi-Agent Systems – SELMAS 2002. (Post-Workshop Report) ACM Software Engineering Notes, August 2002.

[17] A. Garcia, C. Chavez, O. Silva, V. Silva, C. Lucena. “Promoting Advanced Separation of Concerns in Intra-Agent and Inter-Agent Software Engineering”. Workshop on Advanced Separation of Concerns (ASoC) at OOPSLA'2001, Tampa Bay, October 2001

[18] A. Garcia, C. Lucena, D. Cowan. “Agents in Object-Oriented Software Engineering”. Software: Practice and Experience, Elsevier, 2003. (Accepted to Appear)

[19] Garcia, A., Lucena, C., Castro, J., Omicini, A., Zambonelli, F (editors). Software Engineering for Large-Scale Multi-Agent Systems – SELMAS 2002. Lecture Notes in Computer Science, Springer-Verlag, March 2003.

[20] Iglesias, C. et al. "A Survey of Agent-Oriented Methodologies", Proceedings of the ATAL-98, Paris, France, July 1998, pp. 317-330.

[21] Jennings, N., Wooldridge, M. "Agent-Oriented Software Engineering". In: J. Bradshaw (ed.), Handbook of Agent Technology, AAAI/MIT Press, 2000.

[22] E. Kendall, P. Krishna, C. Pathak, C. Suresh, “A Framework for Agent Systems”, in *Implementing Applications Frameworks: Object Oriented Frameworks at Work*, ed. M. Fayad, D. Schmidt, R. Johnson, John Wiley & Sons, 1999.

[23] Kitchenham, B., Pfleeger, S., Fenton, N. Towards a Framework for Software Measurement Validation. IEEE Transactions on Software Engineering 12, 929-944.

[24] Kitchenham B. “Evaluating Software Engineering Methods and Tool”. Part 1 – 12, Software Engineering Notes, vol. 21 no. 1-12, January 1996 - September 1998.

[25] G. Kiczales et al. Aspect-Oriented Programming. European Conference on Object-Oriented Programming (ECOOP), LNCS, (1241), Springer-Verlag, Finland., June 1997.

[26] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. “Getting Started with AspectJ”. Communication of the ACM. October 2001.

[27] Lopes. C. “D: A Language Framework for Distributed Programming" Ph.D. Thesis, College of Computer Science, Northeastern University, 1997.

[28] Meyer. B. Object-Oriented Software Construction, 2nd edition. Prentice Hall, 1997

[29] H. S. Nwana, D. T. Ndumu and L. C. Lee, ZEUS: An advanced Toolkit for Engineering Distributed Multi-Agent Systems, Proceedings of PAAM'98, 1998 (377-391).

[30] Object Management Group – Agent Platform Special Interest Group. “Agent Technology – Green Paper”. Version 1.0, September 2000.
[31] Pace, A., Trilnik, F., Campo, M. Assisting the Development of Aspect-based MAS using the SmartWeaver Approach. In: A. Garcia, C. Lucena, J. Castro, A. Omicini, F. Zambonelli (Eds). "Software Engineering for Large-Scale Multi-Agent Systems". Springer-Verlag, LNCS, March 2003.
[32] Sommerville, I. “Software Engineering”, 6. ed. Harlow, England: Addison-Wesley, 2001.
[33] Silva, V., Garcia, A, Brandao, A., Chavez, C., Lucena, C., Alencar, P. Taming Agents and Objects in Software Engineering. In: "Software Engineering for Large-Scale Multi-Agent Systems". Springer-Verlag, LNCS, March 2003.
[34] Silva, O., Garcia, A, Lucena, C. The Reflective Blackboard Pattern: Architecting Large-Scale Multi-Agent Systems. In: "Software Engineering for Large-Scale Multi-Agent Systems". Springer-Verlag, LNCS, March 2003.
[35] K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa. "The RETSINA MAS Infrastructure." Technical Report CMU-RI-TR-01-05, Robotics Institute Technical Report, Carnegie Mellon, 2001.
[36] L. Sun. “An Experimental Comparison of the Maintainability of Structured Analysis and OO Analysis”.
[37] Tarr, P. et al.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proceedings of the 21st International Conference on Software Engineering, May, 1999.
[38] Wooldridge, M., Jennings, N., and Kinny, D. “The Gaia Methodology for Agent-Oriented Analysis and Design”. In: Autonomous Agents and Multi-Agent Systems, Vol. 3, No. 3, 2000, pp. 285 – 312.
[39] Garcia, A. et al. “Separation of Concerns in Agent-Based Software Engineering: An Empirical Study”. Technical Report 09-03, Computer Science Department, PUC-Rio, February 2003.
[40] E. Kendall. Role Model Designs and Implementations with Aspect-oriented Programming. OOPSLA 1999: 353-369.
[41] Sommerville, I. “Software Engineering”, 5. ed. Harlow, England: Addison-Wesley, 1995.

Appendices A

The use of the GQM methodology generates the following goal and questions:

Goal

Evaluate the reusability and ease of evolution of the implemented multi-agent systems in order to compare the object-oriented development with the aspect-oriented development.

Questions

1. How easy is it to evolve the system?
    1.1. How easy is it to understand the system?
        1.1.1. How concise is the system?
            1.1.1.1. How many components are there?
            1.1.1.2. How many lines of code are there?
            1.1.1.3. How many attributes are there?
            1.1.1.4. How many methods and advices are there?
        1.1.2. How well are the agency concerns localized?
            1.1.2.1. How scattered is the agenthood definition?
            1.1.2.2. How scattered are the agent fundamental properties?
                1.1.2.2.1. How scattered is the autonomy property?
                1.1.2.2.2. How scattered is the interaction property?
                1.1.2.2.3. How scattered is the adaptation property?
            1.1.2.3. How scattered are the agent alternative properties?
                1.1.2.3.1. How scattered is the collaboration property?
            1.1.2.4. How scattered are the agent roles?
            1.1.2.5. How scattered is the definition of agent types and their instances?
                1.1.2.5.1. How scattered is the user agent type definition?
                1.1.2.5.2. How scattered is the definition of a user agent's instance?
                1.1.2.5.3. How scattered is the information agent type definition?
                1.1.2.5.4. How scattered is the definition of an information agent's instance?
        1.1.3. How high is the coupling of the system?
            1.1.3.1. How high is the coupling between components?
        1.1.4. How high is the cohesion of the system?
            1.1.4.1. How high is the cohesion of the system components?
    1.2. How flexible is the system?
        1.2.1. How well are the agency concerns localized?
            1.2.1.1. How scattered is the agenthood definition?
            1.2.1.2. How scattered are the agent fundamental properties?
                1.2.1.2.1. How scattered is the autonomy property?

1.2.1.2.2. How scattered is the interaction property?
1.2.1.2.3. How scattered is the adaptation property?
1.2.1.3. How scattered are the agent alternative properties?
1.2.1.3.1. How scattered is the collaboration property?
1.2.1.4. How scattered are the agent roles?
1.2.1.5. How scattered is the definition of agent types and their instances?
1.2.1.5.1. How scattered is the user agent type definition?
1.2.1.5.2. How scattered is the definition of an user agent's instance?
1.2.1.5.3. How scattered is the information agent type definition?
1.2.1.5.4. How scattered is the definition of an information agent's instance?
1.2.2. How high is the coupling of the system?
1.2.2.1. How high is the coupling between components?
1.2.3. How high is the cohesion of the system?
1.2.3.1. How high is the cohesion of the system components?
1.3. How easy it was to modify the system?
1.3.1. How many changes were made in order to add a new role to some agent type?
1.3.1.1. How many components were added to the system?
1.3.1.2. How many components were changed in the system?
1.3.1.3. How many operations were added to the system?
1.3.1.4. How many operations were changed in the system?
1.3.1.5. How many relationships between components were added to the system?
1.3.1.6. How many relationships between components were changed in the system?
1.3.1.7. How many lines of code were added to the system?
1.3.1.8. How many lines of code were changed in the system?
1.3.2. How many changes were made in order to add a new agent type?
1.3.2.1. How many components were added to the system?
1.3.2.2. How many components were changed in the system?
1.3.2.3. How many operations were added to the system?
1.3.2.4. How many operations were changed in the system?
1.3.2.5. How many relationships between components were added to the system?
1.3.2.6. How many relationships between components were changed in the system?
1.3.2.7. How many lines of code were added to the system?
1.3.2.8. How many lines of code were changed in the system?
1.3.3. How many changes were made in order to add or remove new alternative properties from some agent type?
1.3.3.1. How many components were added to the system?
1.3.3.2. How many components were changed in the system?
1.3.3.3. How many operations were added to the system?
1.3.3.4. How many operations were changed in the system?
1.3.3.5. How many relationships between components were added to the system?
1.3.3.6. How many relationships between components were changed in the system?
1.3.3.7. How many lines of code were added to the system?
1.3.3.8. How many lines of code were changed in the system?
1.3.4. How many changes were made in order to add a new basic property to some agent type, i.e. change the agenthood?
1.3.4.1. How many components were added to the system?

1.3.4.2. How many components were changed in the system?
1.3.4.3. How many operations were added to the system?
1.3.4.4. How many operations were changed in the system?
1.3.4.5. How many relationships between components were added to the system?
1.3.4.6. How many relationships between components were changed in the system?
1.3.4.7. How many lines of code were added to the system?
1.3.4.8. How many lines of code were changed in the system?
1.3.5. How many changes were made in order to add new agent type instances?
1.3.5.1. How many components were added to the system?
1.3.5.2. How many components were changed in the system?
1.3.5.3. How many operations were added to the system?
1.3.5.4. How many operations were changed in the system?
1.3.5.5. How many relationships between components were added to the system?
1.3.5.6. How many relationships between components were changed in the system?
1.3.5.7. How many lines of code were added to the system?
1.3.5.8. How many lines of code were changed in the system?

2. How ease is it to reuse the system elements?
2.1. How easy is it to understand the system?
2.1.1. How concise is the system?
2.1.1.1. How many components are there?
2.1.1.2. How many lines of code are there?
2.1.1.3. How many attributes are there?
2.1.1.4. How many methods and advices are there?
2.1.2. How well are the agency concerns localized?
2.1.2.1. How scattered is the agenthood definition?
2.1.2.2. How scattered are the agent fundamental properties?
2.1.2.2.1. How scattered is the autonomy property?
2.1.2.2.2. How scattered is the interaction property?
2.1.2.2.3. How scattered is the adaptation property?
2.1.2.3. How scattered are the agent alternative properties?
2.1.2.3.1. How scattered is the collaboration property?
2.1.2.4. How scattered are the agent roles?
2.1.2.5. How scattered is the definition of agent types and their instances?
2.1.2.5.1. How scattered is the user agent type definition?
2.1.2.5.2. How scattered is the definition of an user agent's instance?
2.1.2.5.3. How scattered is the information agent type definition?
2.1.2.5.4. How scattered is the definition of an information agent's instance?
2.1.3. How high is the coupling of the system?
2.1.3.1. How high is the coupling between components?
2.1.4. How high is the cohesion of the system?
2.1.4.1. How high is the cohesion of the system components?
2.2. How flexible is the system?
2.2.1. How well are the agency concerns localized?
2.2.1.1. How scattered is the agenthood definition?
2.2.1.2. How scattered are the agent fundamental properties?

2.2.1.2.1. How scattered is the autonomy property?
2.2.1.2.2. How scattered is the interaction property?
2.2.1.2.3. How scattered is the adaptation property?
2.2.1.3. How scattered are the agent alternative properties?
2.2.1.3.1. How scattered is the collaboration property?
2.2.1.4. How scattered are the agent roles?
2.2.1.5. How scattered is the definition of agent types and their instances?
2.2.1.5.1. How scattered is the user agent type definition?
2.2.1.5.2. How scattered is the definition of an user agent's instance?
2.2.1.5.3. How scattered is the information agent type definition?
2.2.1.5.4. How scattered is the definition of an information agent's instance?
2.2.2. How high is the coupling of the system?
2.2.2.1. How high is the coupling between components?
2.2.3. How high is the cohesion of the system?
2.2.3.1. How high is the cohesion of the system components?
2.3. How ease it was to reuse some system elements?
2.3.1. How many changes were made in order to reuse a role in some agent type?
2.3.1.1. How many components were added to the system?
2.3.1.2. How many components were changed in the system?
2.3.1.3. How many operations were added to the system?
2.3.1.4. How many operations were changed in the system?
2.3.1.5. How many relationships between components were added to the system?
2.3.1.6. How many relationships between components were changed in the system?
2.3.1.7. How many lines of code were added to the system?
2.3.1.8. How many lines of code were changed in the system?
2.3.1.9. How many components were copied in the reuse of system elements?
2.3.1.10. How many code lines were copied in the reuse of system elements?
2.3.2. How many changes were made in order to reuse the agenthood while creating a new agent type?
2.3.2.1. How many components were added to the system?
2.3.2.2. How many components were changed in the system?
2.3.2.3. How many operations were added to the system?
2.3.2.4. How many operations were changed in the system?
2.3.2.5. How many relationships between components were added to the system?
2.3.2.6. How many relationships between components were changed in the system?
2.3.2.7. How many lines of code were added to the system?
2.3.2.8. How many lines of code were changed in the system?
2.3.2.9. How many components were copied in the reuse of system elements?
2.3.2.10. How many code lines were copied in the reuse of system elements?
2.3.3. How many changes were made in order to reuse an alternative property in some agent type?
2.3.3.1. How many components (aspects and classes) were added to the system?
2.3.3.2. How many components (aspects and classes) were changed in the system?
2.3.3.3. How many operations (methods and advices) were added to the system?
2.3.3.4. How many operations (methods and advices) were changed in the system?
2.3.3.5. How many relationships between components were added to the system?
2.3.3.6. How many relationships between components were changed in the system?

2.3.3.7. How many lines of code were added to the system?
2.3.3.8. How many lines of code were changed in the system?
2.3.3.9. How many components were copied in the reuse of system elements?
2.3.3.10. How many code lines were copied in the reuse of system elements?

Appendices B

Tables of the metrics results.

General Results

PO Results

Class	CBC	DIT	LOC	LCOM	NOA	WMC
Adaptation	15	2	72		0	9
Answerer	7	5	36	4	1	8
App	19	1	97		0	3
Autonomy	9	2	38	15	1	12
AvailabilityPlan	8	3	16		0	7
Belief	0	1	16	13	1	10
BeliefAgentList	2	2	17	0	1	6
BeliefMyRole	0	2	14	0	1	5
CProposal	0	2	3		0	1
CProposalMsg	1	3	10	0	1	5
Caller	7	5	35	3	1	6
Collaboration	1	3	13		0	14
CollaborationPlan	2	2	6		0	4
CollaborativeAgent	5	2	46	0	1	12
CollaboratorCore	5	4	49	43	2	19
CollaboratorRole	1	4	36	0	1	15
CompositeBelief	1	2	17	0	1	7
CompositeGoal	1	2	17	0	1	7
ContentDistributionPlan	8	3	15		0	7
ContentProposal	0	2	10	0	1	3
ContentSupplier	4	5	16	1	1	4
CoordinatorAgent	3	2	15		0	8
DecisionPlan	3	2	13	1	1	7
EditionGoal	0	2	16	1	2	5
EditionWorkDistributionPlan	6	3	14		0	7
Editor	7	5	24		0	8
Effector	2	1	9	0	1	4
Environment	6	1	62	3	3	12
EnvironmentThread	2	3	36	34	2	22
Goal	1	1	17	15	1	12
GoalMsg	1	2	16	0	1	10
InformationAgent	3	3	35	1	2	8
InformationExchangeGoal	0	2	15	0	1	6
Interaction	7	2	54	36	4	20
MainThread	5	3	46	43	3	22
MakeDecisionGoal	0	2	5		0	1
Message	0	1	25	0	2	10
NegotiationMsg	0	2	5		0	3
NewAgentNotification	1	2	11	0	2	4
Notification	0	1	6		0	2
NotificationMsg	1	2	16	0	1	9

PAgent	12	1	118	50	9	33
Plan	5	1	35	35	3	21
Property	1	1	7		1	1
Proposal	0	1	4		0	1
ProposalMsg	1	3	17	2	2	8
ReactionPlan	2	2	6		0	4
ResourceMsg	0	2	10	0	1	5
ResponseCheckingGoal	0	2	5		0	1
ResponseMsg	1	2	17	2	2	8
ResponseReceivingPlan	8	3	14		0	7
SearchAskAnsweringPlan	6	3	15		0	7
SearchResultReceivingGoal	0	2	12	1	1	4
SearchResultReceivingPlan	9	3	17		0	7
SearchSendPlan	8	3	29		0	9
SearchingGoal	0	2	5		0	1
SearchingPlan	9	3	22		0	7
Sensor	5	1	32	0	4	8
SharedObject	1	1	44	0	4	8
UserAgent	3	3	17	1	1	5

AO Results

Class/Aspect	CBC	DIT	LOC	LCOM	NOA	WMC
Adaptation	15	1	67		0	10
Answerer	8	1	30	6	1	9
App	19	1	97		0	3
Autonomy	9	1	37	15	1	14
AvailabilityPlan	4	3	9		0	7
Belief	0	1	15	13	1	10
BeliefAgentList	2	2	17	0	1	6
BeliefMyRole	0	2	14	0	1	5
Blackboard	6	1	59	3	3	12
CProposal	0	2	3		0	1
CProposalMsg	1	3	10	0	1	5
Caller	9	1	43	10	1	12
Collaboration	8	1	22	4	1	8
CollaborationPlan	2	2	6		0	4
CompositeBelief	1	2	17	0	1	7
CompositeGoal	1	2	17	0	1	7
ContentDistributionPlan	4	3	9		0	7
ContentProposal	0	2	10	0	1	3
ControlThread	2	3	36	34	2	22
ContentSupplier	8	1	19	3	1	7
Coordinator_Agent	3	2	16		0	8
DecisionPlan	3	2	15	1	1	7
EditionGoal	0	2	16	1	2	5
EditionWorkDistributionPlan	6	3	14		0	7
Editor	10	1	29		0	15
Effector	2	1	9	0	1	4
Goal	1	1	17	15	1	12
GoalMsg	1	2	16	0	1	10
InformationExchangeGoal	0	2	15	0	1	6

Information_Agent	3	2	35	1	2	8
Interaction	7	1	59	48	4	25
MainThread	5	3	46	43	3	22
MakeDecisionGoal	0	2	5		0	1
Message	0	1	25	0	2	10
NegotiationMsg	0	2	5		0	3
NewAgentNotification	1	2	11	0	2	4
NewRoleNotification	0	2	9	0	1	3
Notification	0	1	6		0	2
NotificationMsg	1	2	16	0	1	9
PAgent	9	1	95	57	6	29
Plan	5	1	35	35	3	21
Proposal	0	1	4		0	1
ProposalMsg	1	3	17	2	2	8
ReactionPlan	2	2	6		0	4
ResourceMsg	0	2	10	0	1	5
ResponseCheckingGoal	0	2	5		0	1
ResponseMsg	1	2	17	2	2	8
ResponseReceivingPlan	4	3	9		0	7
SearchAskAnsweringPlan	4	3	10		0	7
SearchResultReceivingGoal	0	2	12	1	1	4
SearchResultReceivingPlan	4	3	10		0	7
SearchSendPlan	7	3	29		0	9
SearchingGoal	0	2	5		0	1
SearchingPlan	6	3	13		0	7
Sensor	5	1	32	0	4	8
SharedObject	1	1	44	0	4	8
User_Agent	5	2	17	1	1	5

Totals

CBC

	PO	**AO**
Total	215	196
Max	19	19
Min	0	0
Median	2	2
Average	3.6	3.4

DIT

	PO	**AO**
Total	138	106
Max	5	3
Min	1	1
Median	2	2
Average	2.3	1.9

LOC		
	PO	**AO**
Total	1445	1271
Max	118	97
Min	3	3
Median	16	16
Average	24.1	22.3

LCOM		
	PO	**AO**
Total	304	295
Max	50	57
Min	0	0
Median	1	1
Average	8.2	8.2

NOA		
	PO	**AO**
Total	69	63
Max	9	6
Min	0	0
Median	1	1
Average	1.2	1.1

WMC		
	PO	**AO**
Total	489	460
Max	33	29
Min	1	1
Median	7	7
Average	8.2	8.1

Relevant Differences

Metrics of Size

VS – Vocabulary Size

Project	Total
PO	60
AO	56

LOC – Lines of Code

Class/Aspect	LOC	
	PO Project	AO Project
PAgent	118	95
Interaction	54	59
Collaboration	13	22
Answerer	36	30
Caller	35	43
ContentSupplier	16	19
Editor	24	29
SearchAskAnsweringPlan	15	10
SearchResultReceivingPlan	17	10
SearchingPlan	22	13
AvailabilityPlan	16	9
ContentDistributionPlan	15	9
ResponseReceivingPlan	14	9

Roles and Plans	LOC	
	PO Project	AO Project
Answerer + SearchAskAnsweringPlan	51	40
Caller + SearchingPlan + SearchResultReceivingPlan	74	66
ContentSupplier + AvailabilityPlan	32	28
Editor + ContentDistributionPlan + ResponseReceivingPlan	53	47

Collaboration Property		LOC	
PO Project	Collaboration Class	13	144
	Collaborator Core Class	49	
	Collaborator Role Class	36	
	Collaborative Agent Class	46	
AO Project	Collaboration Aspect	22	22

NOA – Number of Attributes

Class/Aspect	NOA	
	PO Project	AO Project
PAgent	9	6
Collaboration	0	1

Collaboration Property		NOA	
PO Project	Collaboration Class	0	4
	Collaborator Core Class	2	
	Collaborator Role Class	1	
	Collaborative Agent Class	1	
AO Project	Collaboration Aspect	1	1

WMC – Weighted Methods per Component

Class/Aspect	WMC	
	PO Project	AO Project
PAgent	33	29
Adaptation	9	10
Autonomy	12	14
Interaction	17	25
Collaboration	14	12
Answerer	8	9
Caller	6	12
ContentSupplier	4	7
Editor	8	15

Collaboration Property		WMC	
PO Project	Collaboration Class	14	60
	Collaborator Core Class	19	
	Collaborator Role Class	15	
	Collaborative Agent Class	12	
AO Project	Collaboration Aspect	12	12

Metrics of Coupling and Cohesion

CBC - Coupling Between Components

Class/Aspect	CBC	
	PO Project	AO Project
PAgent	12	9
Answerer	7	8
Caller	7	9
ContentSupplier	4	8
Editor	7	10
SearchAskAnsweringPlan	6	4
SearchResultReceivingPlan	9	4
SearchingPlan	9	6
AvailabilityPlan	8	4
ContentDistributionPlan	8	4
ResponseReceivingPlan	8	4

Roles and Plans	CBC	
	PO Project	AO Project
Answerer + SearchAskAnsweringPlan	13	12
Caller + SearchingPlan + SearchResultReceivingPlan	25	19
ContentSupplier + AvailabilityPlan	12	12
Editor + ContentDistributionPlan + ResponseReceivingPlan	23	18

LCOM – Lack of Cohesion in Methods

Class/Aspect	LCOM	
	PO Project	AO Project
PAgent	50	57
Interaction	29	48
Collaboration	-	4
Answerer	4	6
Caller	3	10
ContentSupplier	1	3

DIT – Depth Inherit Tree

Class/Aspect	DIT	
	PO Project	AO Project
Answerer	5	1
Caller	5	1
ContentSupplier	5	1
Editor	5	1

Class/Aspect	DIT	
	PO Project	AO Project
InformationAgent	3	2
UserAgent	3	2

Class/Aspect	DIT	
	PO Project	AO Project
Interaction	2	1
Autonomy	2	1
Adaptation	2	1

Metrics of Separation of Concerns

CDC – Concern Diffusion over Components

Concern	Number of Components	
	PO	AO
Agenthood	25	23
Adaptation	3	1
Answerer	6	1
Autonomy	3	2
Caller	7	1
Collaboration	15	6
Content Supplier	6	1
Editor	7	1
Information Agent	37	33
Instance of Information Agent	3	1
Instance of User Agent	4	2
Interaction	7	6
User Agent	37	33

CDO – Concern Diffusion over Methods

Concern	Number of Methods	
	PO	AO
Agenthood	101	100
Adaptation	10	4
Answerer	21	4
Autonomy	13	6
Caller	21	5
Collaboration	57	30
Content Supplier	19	3
Editor	22	6
Information Agent	162	137
Instance of Information Agent	3	1
Instance of User Agent	4	2
Interaction	29	25
User Agent	156	132

CDLOC – Concern Diffusion over LOC

Concern	Number of Transition Points	
	PO	AO
Agenthood	37	39
Adaptation	13	1
Answerer	13	1
Autonomy	15	3
Caller	15	1
Collaboration	13	1
Content Supplier	13	1
Editor	15	1
Information Agent	41	41
Instance of Information Agent	12	8
Instance of User Agent	14	10
Interaction	25	17
User Agent	35	35